软件测试

# 参考书目

1. Software Testing (2th edition), Ron Patton中译本

大量参考了此书。这本书较薄却设计软件测试的许多方面。虽然有些知识点内容偏少，但条理很清晰。总的来说读起来很轻松。

1. 百度贴吧：<http://tieba.baidu.com/p/4048711941>。  
   发表时间：2015/9/17。访问时间：2015/10/20。

此贴似乎是《Software Testing》某两章内容的总结。

# 软件BUG

## 什么是产品说明书

上网找资料、案例，**深入**了解什么是\*\*\*。

## 如何定义软件BUG

至少出现以下5个规则之一才称存在一个BUG：

1. 未实现产品说明书要求的功能。
2. 出现了产品说明书指明不该出现的错误。
3. 实现了产品说明书未提到的功能。
4. 未实现产品说明书的遗漏之处（有点难理解），如何与第3点做区分？
5. 产品对用户不友好、运行缓慢。

## 为何会出现软件BUG

## 修复BUG的花费

越早修复代价越小，而且代价差距悬殊。

# 软件开发的过程

## 产品的组成部分

### 客户需求

### 产品说明书

### 进度表

### 软件设计文档

### 测试文档

重点了解。

## 软件项目成员

## 软件开发模式

### 大爆炸模式

不写文档，纯粹瞎弄。

### 边写边改模式

几乎没有（或完全没有）计划和文档就直接开始动手写代码。开发流程基本为编码、测试和修复三者的循环过程。等到软件缺陷越来越少，某人就认为是时候发布了。适用于快速制作且用完就扔的小项目，例如：演示程序、产品原型等。

### 瀑布模式

The waterfall model is a sequential design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing, production/implementation and maintenance. [[1]](#footnote-1)



（引自文献1）

瀑布模型就像水流一样自上而下，软件开发过程一步接一步执行。分为产品构思、需求分析、产品设计、开发、测试和发布维护6个步骤。无法回溯，一旦进入一个步骤就不能修改上一步骤。无法回溯意味着每一阶段都已有明确的产品定义和开发方向，是该模型巨大的优点（如果产品定义明确）。但在这个迅速变化的互联网时代，无法回溯也是巨大的缺点（需求和设计经常发生变化）。

The waterfall development model originates in the manufacturing and construction industries: highly structured physical environments in which after-the-fact changes are prohibitively costly, if not impossible. Since no formal software development methodologies existed at the time, this hardware-oriented model was simply adapted for software development.[[2]](#footnote-2)

瀑布模型起源于制造业和建筑业：改变原先的计划将导致昂贵的代价（或者说根本不可能）。因为那时候还没有正式的软件开发方法学，瀑布模型才被用在软件开发。

### 螺旋模式

结合了瀑布模式、边写边改模式和大爆炸模式。可以算是不错的软件开发模式。对于软件测试人员也很友好，因为可以在最初阶段就开始参与测试从而尽早发现软件缺陷。

### 敏捷软件开发

很容易偏离主题，对程序员要求较高。

# 不要追求完美

## 不可能完全测试程序

如果能够完全测试程序那么就可以找出软件的所有BUG，但这是不可能的。因为输入量太大。即便是一个最简单的计算器程序，完全测试需要测试：

* 1+1
* 1+2
* 一直加下去…
* 1+99999999999
* 99999999999+99999999999

然而这仅仅是正整数加法。

## 软测的实质

软件测试可以证明软件存在BUG（只要找到一个）。但软件测试无法证明程序没有BUG，除非进行完全测试（但这是不可能的）。因此不存在完美的程序，软件测试的思想是利用有限的精力测试最重要的部件。

## 最佳测试量

测试会不断发现软件BUG，理想的情况是让测试永远进行下去。但这是不可能的，因为测试带来的费用不可小视而且受到项目进度的限制。测试不足会导致软件缺陷非常多，而过度测试导致高昂的费用。我们需要找到一个合适的测试量。

（图：测试费用与软件收益）

## 有些BUG可以不修复

项目小组需要进行取舍，有些BUG可以放弃修复。主要原因有：

* 项目赶得紧，没时间。通常还要结合BUG的严重性做出决策。
* 找到的BUG不算真正的BUG。可能是测试人员不理解软件缺陷的定义。也可能是测试人员不理解产品。因此产品说明书的质量是很重要的。
* 修复的风险太大。这点很常见，对程序做出修改常常会导致更多的BUG。
* 不值得修复。可能BUG出现在不常用的部件中。也可能用户有其他方法避开此BUG。

可以放弃修复某些BUG，但错误的决策可能导致严重的后果。1994年因特尔奔腾处理器浮点除法运算存在某个BUG。因特尔在产品发布前就了解到该BUG，但没有选择修复，最终因特尔被迫花费4亿多美元更换有问题的芯片。

# 测试前的准备

## 一些术语

### 黑盒测试、白盒测试

根据能否看到源代码区分。介于两者之间还有灰盒测试。

### 静态测试、动态测试

根据测试是否需要运行软件区分。

### 软件测试、质量保证

## 常见误区

## 测试心理学

### 谁不能参与软件测试

## 测试工具

# 检查产品说明书（静态黑盒测试）

大爆炸模式没有产品说明书，其它模式或多或少都有产品说明书。

* 幸运！项目早期测试产品说明书。如果能够在这时测试、修改产品说明书是极好的情况。能够在此阶段找出软件缺陷能够节省大量开销和时间。
* 不幸！软测人员半途加入，此时产品说明书早已写完，编码工作也已经持续了很久。但你还是有必要检查产品说明书。

# 动态黑盒测试

## 什么是动态黑盒测试

## 通过性测试和失效性测试

## 测试技巧

### 等价类划分

我们提到过不可能穷举所有可能的输入。等价类划分的意义在于它可以大大缩小测试用例的数量。

对于计算器应用，来看下面的测试用例。

|  |  |  |
| --- | --- | --- |
| 编号 | 用例 | 预期结果 |
| 1 | 2+3 | 5 |
| 2 | 3+4 | 7 |
| 3 | 1+65535 | 65536 |

2+3测试通过后还需要测试3+4吗？如果你有些编程经验，当然不必测试3+4了。那1+65535呢？有经验的程序员应该意识到可能存在溢出错误。那么我们说1号用例和2号用例属于同一个等价类，不一定要全部测试。

### 数据测试

#### 边界值（通过性测试）

边界值可以作为等价类划分的依据。

#### 测试边界（通过性测试）

#### 空值（通过性测试）

用户可能在登录的时候忘记输入用户名。测试时应将空值作为单独的等价类。

#### 非法数据（失效性测试）

不要总期待用户会用正确的方式使用你的软件。总有些“千奇百怪”的用户：

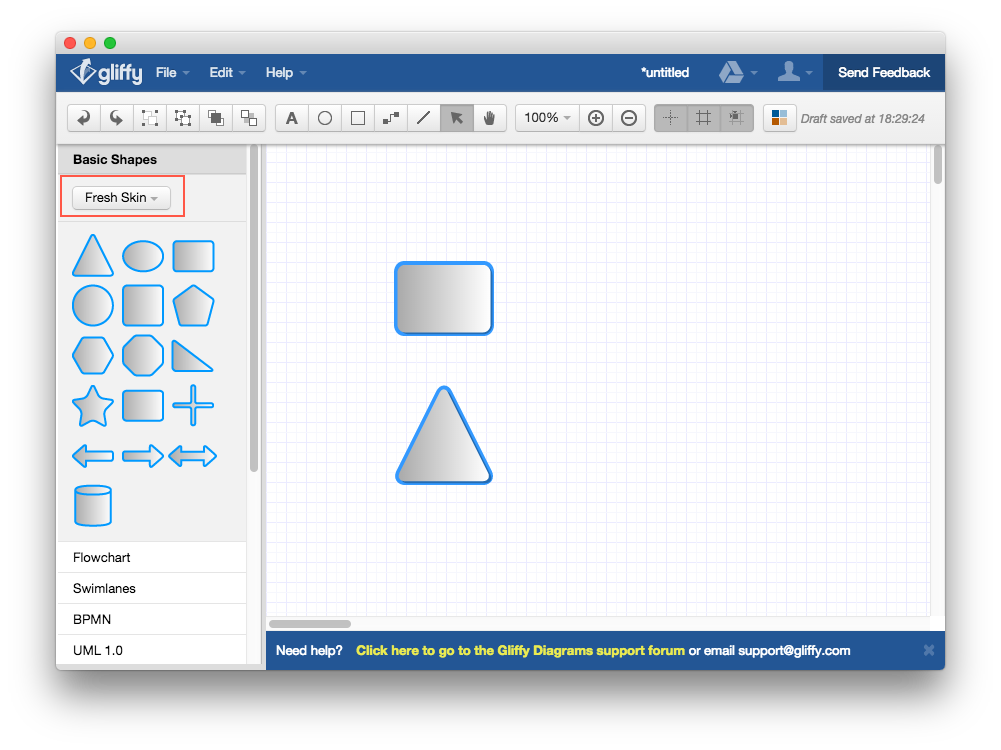
1. 在计算器应用中输入字母
2. 输入负数作为账户余额
3. 将时间设置在公元3000年

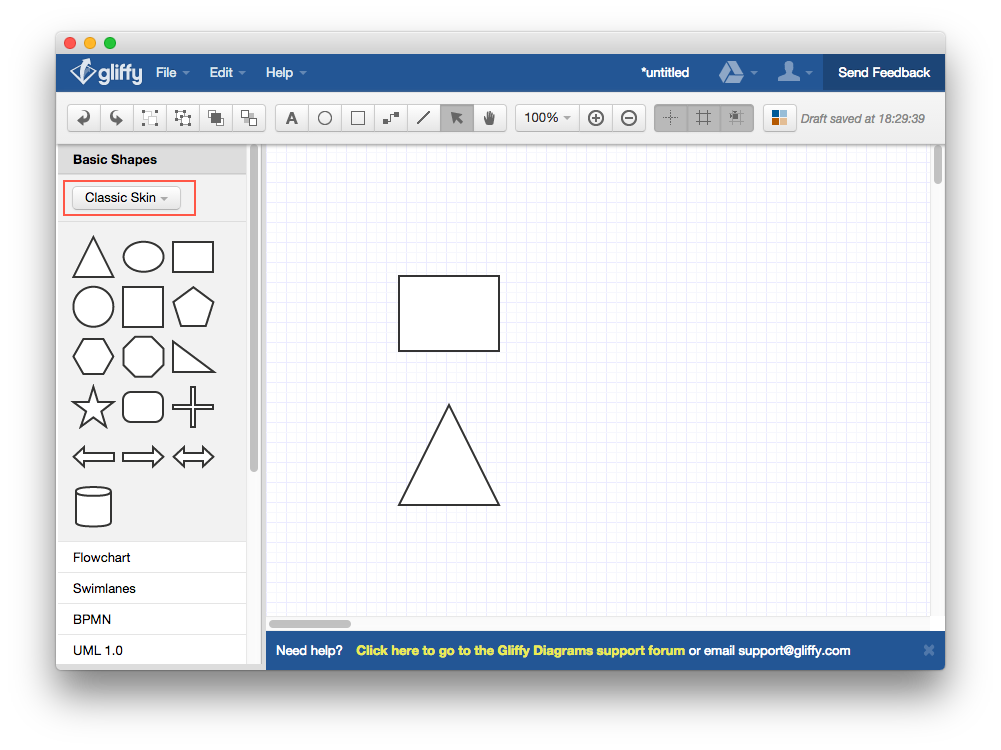
### 软件状态

状态是指软件当前所处的模式。用户可能从一种状态切换到另一种状态、再切换到另一种状态、再切换…再切换…。对于绘图应用而言，用户可能有如下状态切换流程：

|  |  |  |
| --- | --- | --- |
| 画笔颜色 | 画笔类型 | 备注 |
| 红色 | 铅笔 | 初始状态 |
| 红色 | 刷子 | 换画笔 |
| 绿色 | 刷子 | 换颜色 |
| 黑色 | 马克笔 | 同时换画笔和画刷 |

下面介绍一个Chrome应用：gliffy，可以绘制流程图、UML等。图\*和图\*展现了gliffy所处的两种不同状态：经典皮肤和新皮肤。





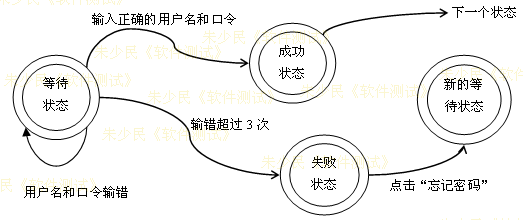
### 通过性状态测试

#### 为什么不能测试全部状态

如果有5种状态，那么走遍他们有120种走法，找到最短路径还算容易（但实际软件可能有几百种状态）。这部分我没理解：软件有5种状态，那么需要测试5种情况还是测试120种情况？

#### 建立状态转换图

有好几种绘图方式，只要项目组看得懂就可以。状态转换图可能非常大，那就只能借助商业软件绘制。



#### 减少状态及状态转换的数量

#### 常被忽略的软件状态

### 失效性状态测试

#### 竞争条件和时序错乱

#### repetition testing、stress testing和load testing

# 静态白盒测试

# 动态白盒测试

1. Waterfall model. (2015, October 16). Retrieved October 18, 2015, from Wikipedia:

   https://en.wikipedia.org/wiki/Waterfall\_model [↑](#footnote-ref-1)
2. Benington H D. Production of large computer programs[J]. IEEE Annals of the History of Computing, 1983 (4): 350-361. [↑](#footnote-ref-2)