[一、冒烟测试 2](#_Toc5503)

[1.1简介 2](#_Toc17526)

[1.2特点 3](#_Toc30081)

[1.3优缺点 3](#_Toc2043)

[1.4应用场景 4](#_Toc359)

[1.5准则 5](#_Toc23942)

[1.5.1、与开发人员协同工作 5](#_Toc7987)

[1.5.2、在进行冒烟测试前检查代码  5](#_Toc2686)

[1.5.3、在干净的调试版本中安装私有二进制文件  6](#_Toc24662)

[1.5.4、创建每日构建  6](#_Toc23955)

[1.5.5、Web测试和负载测试  6](#_Toc22460)

[1.6结合项目说明 7](#_Toc2324)

[二、功能测试 7](#_Toc30018)

[2.1简介 7](#_Toc20749)

[2.2特点 7](#_Toc14811)

[2.3优缺点 8](#_Toc13139)

[2.3.1、功能测试优点 8](#_Toc29172)

[2.3.2、功能测试缺点 8](#_Toc8418)

[2.4测试用例方法 9](#_Toc20436)

[2.4.1、等价类划分法 9](#_Toc23355)

[2.4.2、边界值分析法 11](#_Toc19971)

[2.4.3、错误推测法 12](#_Toc23970)

[2.4.4、因果图法 12](#_Toc16906)

[2.4.5、判定表组成法 13](#_Toc14708)

[2.5测试流程 15](#_Toc2563)

[2.5.1、测试计划 15](#_Toc17890)

[2.5.2、测试设计 15](#_Toc21456)

[2.5.3、测试开发 15](#_Toc18135)

[2.5.4、测试执行 15](#_Toc13050)

[2.5.5、测试评估 15](#_Toc3038)

[2.6集合项目说明 16](#_Toc8829)

[三、回归测试 16](#_Toc20272)

[3.1简介 16](#_Toc14867)

[3.2观念 16](#_Toc7778)

[3.3测试策略 17](#_Toc23230)

[3.3.1测试用例库的维护 17](#_Toc8514)

[3.3.2回归测试包的选择 19](#_Toc29847)

[3.4测试流程 21](#_Toc30954)

[3.5集合项目说明 21](#_Toc22577)

软件表面上看起来的一些功能和内容，通常要考靠有计划和条理的开发过程来实现，那么在这个从开始到计划，到编程，到测试，到最后公开使用的过程中，都有可能发现软件的BUG（缺陷）。有研究表明，修复软件BUG的费用是呈指数级增加的，图1清楚的显示了BUG发现的时间是和修复的费用成正比增长的。

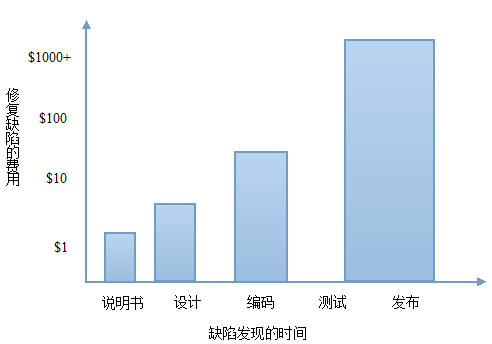


图1

由此可见，软件缺陷发现的越早付出的代价就越小，尽可能的在软件发布之前，尽量找出所有能被发现的软件缺陷，这是对软件后期的稳定运行的保证。软件测试对于一个软件工程来说是很重要的，对软件质量的好与否有着不可或缺的影响。接下来我将论述冒烟测试、功能测试、回归测试这三种测试方法，也是在软件测试中常用的几种方法。软件测试的测试人员一般从分析需求开始，然后根据自己的测试能力和经验来制定测试计划，对于一个软件的测试来说，这三种测试方法的顺序可以按照冒烟测试、功能测试、回归测试来进行。

**一、冒烟测试**

**1.1简介**

关于冒烟测试，应该是微软首先提出来的一个概念，和微软一直提倡的每日build有很密切的联系。具体说，冒烟测试就是在每日build建立后，对系统的基本功能进行简单的测试。这种测试强调功能的覆盖率，而不对功能的正确性进行验证。从这一点看和所谓的“接受性（验收）测试（Acceptance Test）”非常相似。不同之处就在于他们执行的频率和被测的版本不同。

至于冒烟测试这个名称的来历，大概是从电路板测试得来的。因为当电路板做好以后，首先会加电测试，如果板子没有冒烟在进行其它测试，否则就必须重新来过。类似的如果冒烟测试没有通过，那么这个build也会返回给开发队伍进行修正，测试人员测试的版本必须首先通过冒烟测试的考验。

在软件中，“冒烟测试”这一术语描述的是在将代码更改嵌入到产品的源树中之前对这些更改进行验证的过程。

在测试中发现问题，找到了一个Bug，然后开发人员会来修复这个Bug，这时想知道这次修复是否真的解决了程序的Bug或者是否会对其它模块造成影响，就需要针对此问题进行专门测试，这个过程就被称为冒烟测试（Smoke Test）。

**1.2特点**

冒烟测试，确认软件基本的功能正常，保证软件系统能跑的起来，可以进行后续的正式测试工作，如果基本的测试都有问题，就直接打回开发部了，所以正式交付测试的版本前必须首先通过冒烟测试的考验。

冒烟测试只是一个测试活动，并不是一个测试阶段，也就是说，冒烟测试贯穿于测试的任何一个阶段。也就是在正式测试前，对系统的基本功能进行简单的测试，这种测试强调功能的覆盖率，而不对功能的正确性进行验证。

在检查代码之后，冒烟测试是确定和修复软件缺陷的最经济有效的方法。冒烟测试设计用于确认代码中的更改会按预期运行，且不会破坏整个版本的稳定性。冒烟测试是自由测试中的一种。

冒烟测试的名称可以理解为该种测试耗时短，仅用一袋烟功夫足够了。也有人认为是形象地类比新电路板基本功能检查。任何新电路板焊好后，先通电检查，如果存在设计缺陷，电路板可能会短路，板子冒烟了。

**1.3优缺点**

1.3.1冒烟测试优点

（1）进度可见并可以控制到1-2天的细粒度，很容易看到进度的偏差；

（2）及早的发现开发BUG和缺陷并分析解决，对开发人员的一种监督和促进，提高软件质量 ；

（3）由于将大集成分解到每日构建中的小集成，避免了传统产品集成或集成测试时候出现的严重问题的可能；

（4）在项目中倡导质量意识，强调第一次就把事情做好，而不是等后期才发现问题。

1.3.2冒烟测试缺陷

（1）给开发人员太大压力，开发每天都在较紧张环境中工作 ；

（2）需要额外的测试人力资源和每日构建硬件环境的投入 ；

（3）开发人员不能专注，既要分心去修改BUG，又要开发新的功能点；

（4）对开发负责人要求更好，需要将功能细化到1-2天的有明确输出的功能点

（5）开发需要投入额外的精力来保证每日构建顺畅 。

**1.4应用场景**

冒烟测试的对象是每一个新编译的需要正式测试的软件版本，目的是确认软件基本功能正常，可以进行后续的正式测试工作。冒烟测试的执行者是版本编译人员。

在软件公司，软件在编写过程中，内部需要编译多个版本，但是只有有限的几个版本需要执行正式测试，这些需要执行的中间测试版本，在刚刚编译出来后，软件编译人员需要进行基本性能确认测试，例如是否可以正确安装/卸载，主要功能是否实现，是否存在严重死机或数据严重丢失等Bug。如果通过了该测试，则可以根据正式测试文档进行正式测试。否则，就需要重新编译版本，再次执行版本可接收确认测试，直到成功。

主要应用在以下的场景中：

（1）对进度偏差控制和要求很高的项目；（2）开发检查点和里程碑制定的很细致的项目；（3）采用增量和迭代开发的项目,快速和敏捷开发的项目。

**1.5准则**

下面的准则描述了冒烟测试的最佳做法。遵循准则的效果会有很大的不同，从增强团队成员之间的交流，到形成特定的使用测试和调试工具的方式等。

1.5.1、与开发人员协同工作

由于冒烟测试特别关注更改过的代码，因此必须与编写代码的开发人员协同工作。必须了解以下内容：

（1）代码中进行了什么更改。若要理解该更改，必须理解使用的技术；开发人员可以提供相关说明。

（2）更改对功能有何影响。

（3）更改对各组件的依存关系有何影响。

1.5.2、在进行冒烟测试前检查代码

在运行冒烟测试前，进行侧重于代码中的所有更改的代码检查。代码检查是验证代码质量并确保代码无缺陷和错误的最有效、最经济的方法。冒烟测试确保通过代码检查或风险评估标识的主要的关键区域或薄弱区域已通过验证，因为如果失败，测试就无法继续。

1.5.3、在干净的调试版本中安装私有二进制文件

由于冒烟测试必须侧重于仅对更新后的二进制文件中的功能更改进行验证，所以必须通过使用被测试文件的调试二进制文件来使测试在干净的测试环境中运行。

1.5.4、创建每日构建

每日构建要求团队成员协同工作，并鼓励开发人员彼此保持同步。如果新版本的迭代被延迟，则该延迟很容易导致具有多个依赖项的产品不同步。遵循每日构建和冒烟测试的过程，任何更改过的或新的二进制文件都可确保实现高质量。

将高质量的每日构建作为团队最重要的任务。如果由于签入代码未进行冒烟测试而导致版本中断，则需要开发人员和测试人员停止所有其他工作，直到问题被解决为止。对导致中断版本的人员的处罚不应该很重，但这个处罚一定要能强调这样一个道理：正确的每日构建是团队最重要的任务。

不需要执行穷举测试。冒烟测试的目的不是确保二进制文件100%没有错误。这样需要花费太多的时间。执行冒烟测试是为了在高级别验证版本。要确保二进制文件中的更改不会破坏常规版本的稳定性，也不会导致功能中出现严重错误。

1.5.5、Web测试和负载测试

生成Web测试和负载测试时，在运行任何时间长、工作量大的测试之前运行冒烟测试是一种很好的做法。在Web测试和负载测试中，冒烟测试时间短，工作量也小。使用冒烟测试是为了在运行性能测试或压力测试之前，确保一切都已正确配置并可按预期运行。

**1.6结合项目说明**

项目一：

Auth为authorization的简称。这个身份验证应用程序（auth application）允许添加、更新和查询外部身份验证。每个独特的成员有着不同的ID，只有已授权的成员才能进行信息的增删查改。

auth maintenance history是对AUTH的增删查改历史信息的记录，在这里可以看到所有AUTH的详细信息。

我在测试auth maintenance history的enhancement的时候，我会将新需求文档和原来的需求文档进行对比，把需求变化的点整理成一份新的文档。接下来，先将变化的功能点（Preauth、Customized auth、Restricted auth下的树形结构（如图2）里的内容和Auth history table的fields（如图3）测试一下，如果树形结构能正常显示，并且点击每个Auth type下有值的链接能出来相应的history table，就可以说冒烟测试成功。

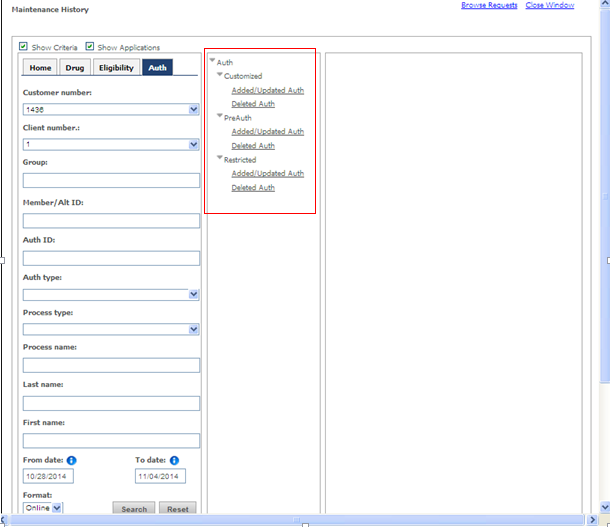


图2

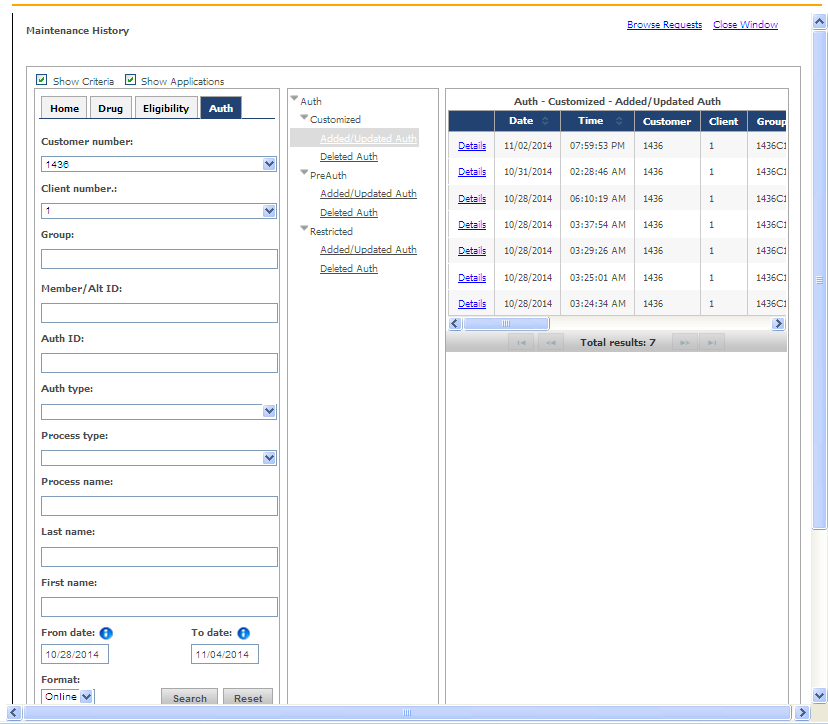


图3

项目二：

Security是对登陆系统的用户权限的的安全设置。每个用户拥有不同的权限，data role,application role,job function role,

在这里所要举的例子是有关job function的，job function部分增加了一个Owner域，那作为一个测试人员，我肯定要知道在没改动之前的job function是什么样的，job function的history页面是什么样的。在看job function这部分原来的系统页面的同时，需要我去看这部分原来的需求文档，把需求文档弄清楚之后，会根据变动，将要测试的内容写成一份测试用例（），等开发在服务器上部署好了job function Security之后, 我会根据测试用例把这个项目#3650 进行下smoke test， 看基本功能能否跑通（主要是下面4点内容）

主要的改动是Job function 页面加了Owner这一个域；需要测试的是：

（1）Job function 页面的CRUD（增删查改）；

（2）Export / Import – Job functions相关内容；

（3）Maintenance History – Job function相关；

（4）将新建的job function加到一个user中，看能否起作用；

这四个点看起来很简单，但其实也需要一系列衔接的操作和页面反应。然后根据这四点相关的测试用例的测试通过率来评判冒烟测试是否成功。以下这几种情况的出现，则可以认为冒烟测试没有成功：当Job function 页面CRUD（增删查改）的时候，没有Owner这个域出现；也或者如果因为Job function 页面加了Owner这一个域而使得用户的权限变得不受控制，即当用户登陆的时候没有出现相应的权限可以操作的内容；使得MH页面的搜索不出结果，或者搜索出来的结果不符合搜索条件；Export / Import 一些Job functions时，Owner域的值没有显示出来。

**二、功能测试**

**2.1简介**

功能测试（Functional testing），也称为行为测试（behavioral testing）、黑盒测试（Black-box Testing），根据产品特性、操作描述和用户方案，测试一个产品的特性和可操作行为以确定它们满足设计需求。在测试中，把程序看作一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息。

本地化软件的功能测试，用于验证应用程序或网站对目标用户能正确工作。使用适当的平台、浏览器和测试脚本，以保证目标用户的体验将足够好，就像应用程序是专门为该市场开发的一样。功能测试是为了确保程序以期望的方式运行而按功能要求对软件进行的测试，通过对一个系统的所有的特性和功能都进行测试确保符合需求和规范。

**2.2特点**

功能测试只需考虑需要测试的各个功能，着眼于程序外部结构，不需要考虑整个软件的内部结构及代码。一般从软件产品的界面、架构出发，按照需求编写出来的测试用例，输入数据在预期结果和实际结果之间进行评测，进而提出更加使产品达到用户使用的要求。该测试方法是穷举输入测试，只有把所有可能的输入都作为测试情况使用，才能以这种方法查出程序中所有的错误。实际上测试情况有无穷多个，人们不仅要测试所有合法的输入，而且还要对那些不合法但是可能的输入进行测试。

主要试图发现下列几类错误：功能不正确或遗漏；界面错误；输入和输出错误；数据库访问错误；性能错误；初始化和终止错误等。

**2.3优缺点**

2.3.1、功能测试优点

（1）基本上不用人管着，如果程序停止运行了一般就是被测试程序宕了；

（2）对比较大的代码单元来说，黑盒测试比白盒测试效率要高；

（3）测试人员不需要了解实现的细节，包括特定的编程语言；

（4）测试人员和编码人员是彼此独立的；

（5）从用户的视角进行测试，很容易被理解和接受；

（6）有助于暴露任何规格不一致或者有歧义的问题；

（7）测试用例可以在规格完成之后马上执行。

2.3.2、功能测试缺点

（1）测试的只有一小部分，不可能测试全部输入；

（2）没有清洁和简明的需求规格说明书，测试用例很难设计；

（3）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；

（4）很多测试路径没有测试到；

（5）不能直接对特定程序段进行测试，改程序段可能隐藏更多错误；

（6）大部分和研究相关的测试都是直接针对白盒测试的。

**2.4测试用例方法**

从理论上讲，功能测试只有采用穷举输入测试，把所有可能的输入都作为测试情况考虑，才能查出程序中所有的错误。实际上测试情况有无穷多个，人们不仅要测试所有合法的输入，而且还要对那些不合法但可能的输入进行测试。这样看来，完全测试是不可能的，所以我们要进行有针对性的测试，通过制定测试案例指导测试的实施，保证软件测试有组织、按步骤，以及有计划地进行。测试行为必须能够加以量化，才能真正保证软件质量，而测试用例就是将测试行为具体量化的方法之一。具体的黑盒测试用例设计方法包括等价类划分法、边界值分析法、错误推测法、因果图法、判定表驱动法、正交试验设计法、功能图法、场景法等。

2.4.1、等价类划分法

等价类划分的办法是把程序的输入域划分成若干部分（子集），然后从每个部分中选取少数代表性数据作为测试用例。每一类的代表性数据在测试中的作用等价于这一类中的其他值。该方法是一种重要的，常用的黑盒测试用例设计方法。

（1）划分等价类：等价类是指某个输入域的子集合。在该子集合中，各个输入数据对于揭露程序中的错误都是等效的，并合理地假定：测试某等价类的代表值就等于对这一类其它值的测试。因此，可以把全部输入数据合理划分为若干等价类，在每一个等价类中取一个数据作为测试的输入条件，就可以用少量代表性的测试数据。取得较好的测试结果。等价类划分可有两种不同的情况：有效等价类和无效等价类。

有效等价类：是指对于程序的规格说明来说是合理的，有意义的输入数据构成的集合。利用有效等价类可检验程序是否实现了规格说明中所规定的功能和性能。

无效等价类：与有效等价类的定义恰巧相反。

设计测试用例时，要同时考虑这两种等价类。因为，软件不仅要能接收合理的数据，也要能经受意外的考验。这样的测试才能确保软件具有更高的可靠性。

（2）划分等价类的方法：下面给出六条确定等价类的原则：

1）在输入条件规定了取值范围或值的个数的情况下，则可以确立一个有效等价类和两个无效等价类。

2）在输入条件规定了输入值的集合或者规定了“必须如何”的条件的情况下，可确立一个有效等价类和一个无效等价类。

3）在输入条件是一个布尔量的情况下，可确定一个有效等价类和一个无效等价类。

4）在规定了输入数据的一组值（假定n个），并且程序要对每一个输入值分别处理的情况下，可确立n个有效等价类和一个无效等价类。

5）在规定了输入数据必须遵守的规则的情况下，可确立一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

6）在确知已划分的等价类中各元素在程序处理中的方式不同的情况下，则应再将该等价类进一步的划分为更小的等价类。

2.4.2、边界值分析法

边界值分析是通过选择等价类边界的测试用例。边界值分析法不仅重视输入条件边界，而且也必须考虑输出域边界。它是对等价类划分方法的补充。

（1）边界值分析方法的考虑：

大量的错误是发生在输入或输出范围的边界上，而不是发生在输入输出范围的内部。因此针对各种边界情况设计测试用例，可以查出更多的错误。

使用边界值分析方法设计测试用例，首先应确定边界情况。通常输入和输出等价类的边界，就是应着重测试的边界情况。应当选取正好等于，刚刚大于或刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值作为测试数据。

（2）基于边界值分析方法选择测试用例的原则：

1）如果输入条件规定了值的范围，则应取刚达到这个范围的边界的值，以及刚刚超越这个范围边界的值作为测试输入数据。

2）如果输入条件规定了值的个数，则用最大个数,最小个数，比最小个数少一，比最大个数多一的数作为测试数据。

3）根据规格说明的每个输出条件，使用前面的原则1）。

4）根据规格说明的每个输出条件，应用前面的原则2）。

5）如果程序的规格说明给出的输入域或输出域是有序集合，则应选取集合的第一个元素和最后一个元素作为测试用例。

6）如果程序中使用了一个内部数据结构，则应当选择这个内部数据结构的边界上的值作为测试用例。

7）分析规格说明，找出其它可能的边界条件。

2.4.3、错误推测法

错误推测法是基于经验和直觉推测程序中所有可能存在的各种错误，从而有针对性的设计测试用例的方法。

错误推测方法的基本思想： 列举出程序中所有可能有的错误和容易发生错误的特殊情况，根据他们选择测试用例。 例如，在单元测试时曾列出的许多在模块中常见的错误。 以前产品测试中曾经发现的错误等，这些就是经验的总结。还有，输入数据和输出数据为0的情况。 输入表格为空格或输入表格只有一行。 这些都是容易发生错误的情况。可选择这些情况下的例子作为测试用例。

2.4.4、因果图法

前面介绍的等价类划分方法和边界值分析方法，都是着重考虑输入条件，但未考虑输入条件之间的联系，相互组合等。 考虑输入条件之间的相互组合，可能会产生一些新的情况。 但要检查输入条件的组合不是一件容易的事情，即使把所有输入条件划分成等价类，他们之间的组合情况也相当多。 因此必须考虑采用一种适合于描述对于多种条件的组合，相应产生多个动作的形式来考虑设计测试用例。 这就需要利用因果图（逻辑模型）。

因果图方法最终生成的就是判定表。它适合于检查程序输入条件的各种组合情况。

（1）生成测试用例

1）分析软件规格说明描述中，哪些是原因（即输入条件或输入条件的等价类），哪些是结果（即输出条件），并给每个原因和结果赋予一个标识符。

2）分析软件规格说明描述中的语义。找出原因与结果之间，原因与原因之间对应的关系。 根据这些关系，画出因果图。

3）由于语法或环境限制，有些原因与原因之间，原因与结果之间的组合情况不可能出现。 为表明这些特殊情况，在因果图上用一些记号标明约束或限制条件。

4）把因果图转换为判定表。

5）把判定表的每一列拿出来作为依据，设计测试用例。

从因果图生成的测试用例（局部，组合关系下的）包括了所有输入数据的取TRUE与取FALSE的情况，构成的测试用例数目达到最少，且测试用例数目随输入数据数目的增加而线性地增加。

2.4.5、判定表组成法

判定表（Decision Table）是分析和表达多逻辑条件下执行不同操作的情况下的工具。在程序设计发展的初期，判定表就已被当作编写程序的辅助工具了。由于它可以把复杂的逻辑关系和多种条件组合的情况表达得既具体又明确。

条件桩（Condition Stub）：列出了问题的所有条件。通常认为列出的条件的次序无关紧要。

动作桩（Action Stub）：列出了问题规定可能采取的操作。这些操作的排列顺序没有约束。

条件项（Condition Entry）：列出针对它左列条件的取值。在所有可能情况下的真假值。

动作项（Action Entry）：列出在条件项的各种取值情况下应该采取的动作。

规则：任何一个条件组合的特定取值及其相应要执行的操作。在判定表中贯穿条件项和动作项的一列就是一条规则。显然，判定表中列出多少组条件取值，也就有多少条规则，既条件项和动作项有多少列。

（1）判定表的建立步骤

1）确定规则的个数。假如有n个条件。每个条件有两个取值（0,1），故有2n种规则。

2）列出所有的条件桩和动作桩。

3）填入条件项。

4）填入动作项。等到初始判定表。

5）简化。合并相似规则（相同动作）。

（2）判定表设计测试用例的条件

1）规格说明以判定表形式给出，或很容易转换成判定表。

2）条件的排列顺序不会也不影响执行哪些操作。

3）规则的排列顺序不会也不影响执行哪些操作。

4）每当某一规则的条件已经满足，并确定要执行的操作后，不必检验别的规则。

5）如果某一规则得到满足要执行多个操作，这些操作的执行顺序无关紧要。

**2.5测试流程**

2.5.1、测试计划

首先，根据用户需求报告中关于功能要求和性能指标的规格说明书，定义相应的测试需求报告，即制订黑盒测试的最高标准，以后所有的测试工作都将围绕着测试需求来进行，符合测试需求的应用程序即是合格的，反之即是不合格的；同时，还要适当选择测试内容，合理安排测试人员、测试时间及测试资源等。

2.5.2、测试设计

将测试计划阶段制订的测试需求分解、细化为若干个可执行的测试过程，并为每个测试过程选择适当的测试用例（测试用例选择的好坏将直接影响到测试结果的有效性）。

2.5.3、测试开发

建立可重复使用的自动测试过程。

2.5.4、测试执行

执行测试开发阶段建立的自动测试过程，并对所发现的缺陷进行跟踪管理。测试执行一般由单元测试、组合测试、集成测试、系统联调及回归测试等步骤组成，测试人员应本着科学负责的态度，一步一个脚印地进行测试。

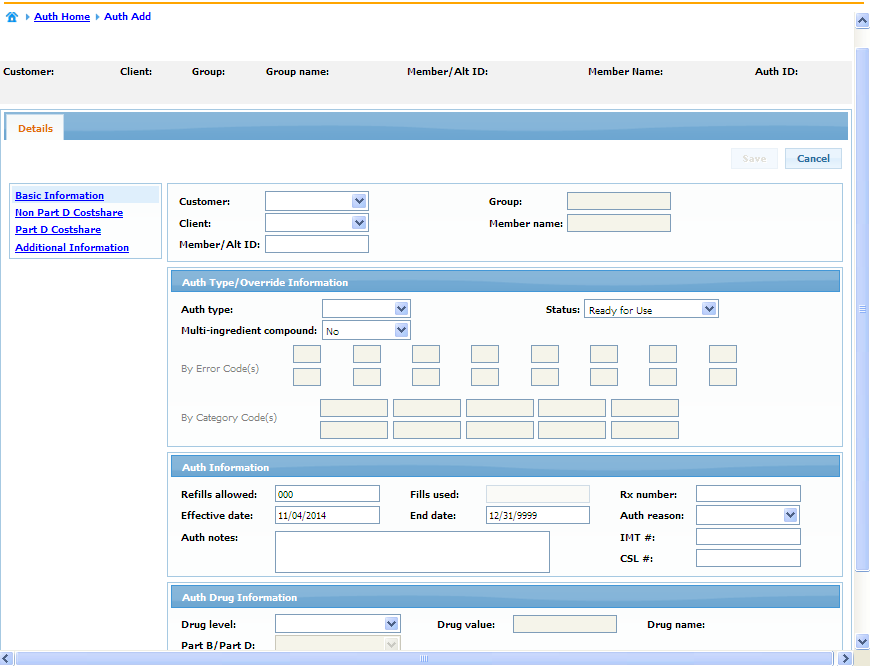
2.5.5、测试评估

结合量化的测试覆盖域及缺陷跟踪报告，对于应用软件的质量和开发团队的工作进度及工作效率进行综合评价。

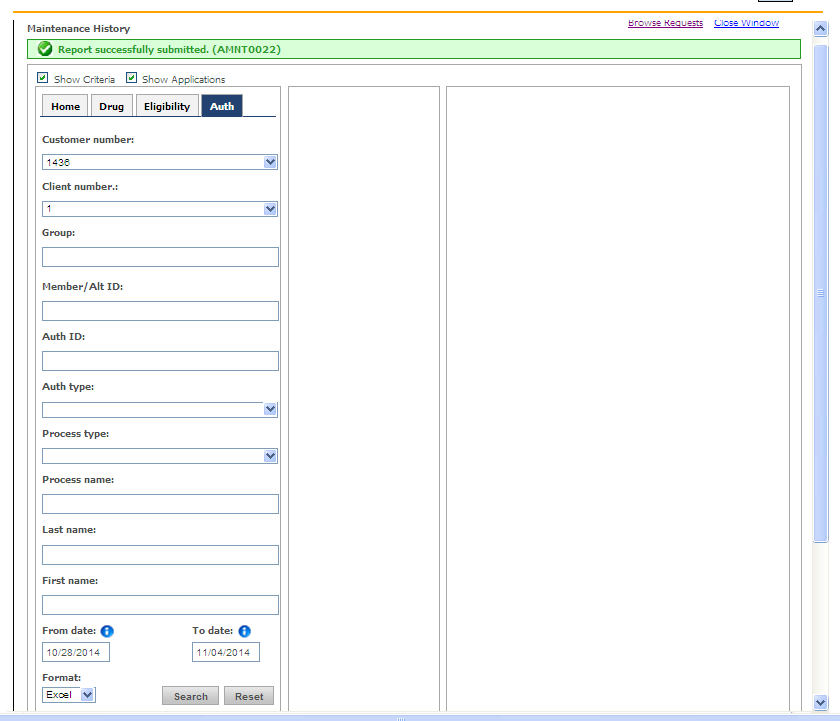
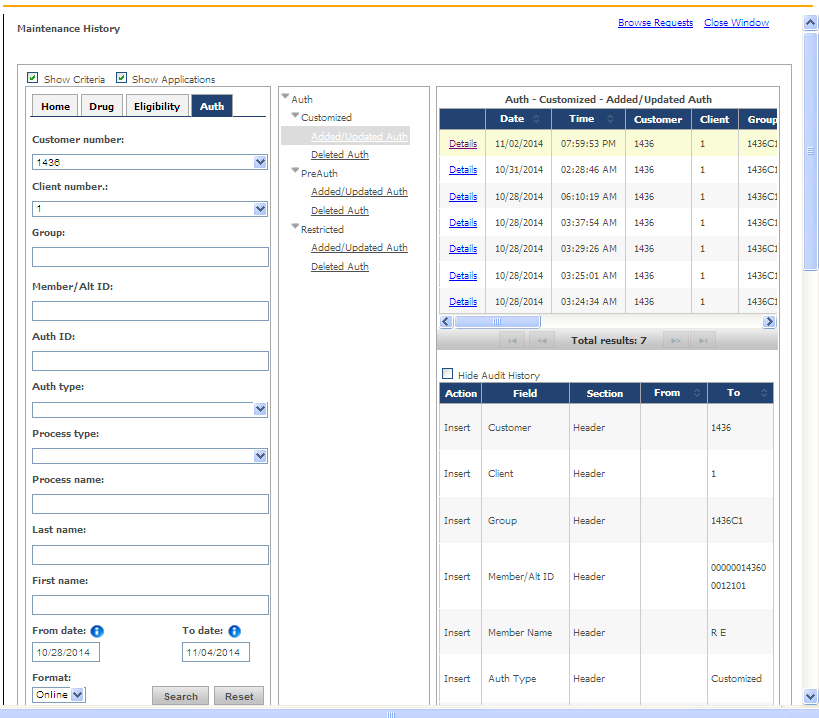
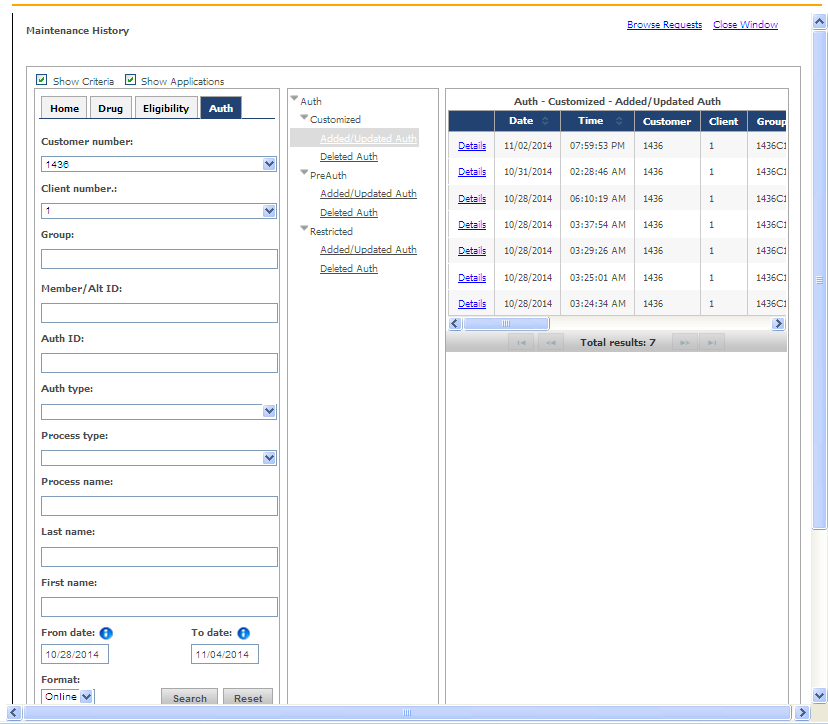
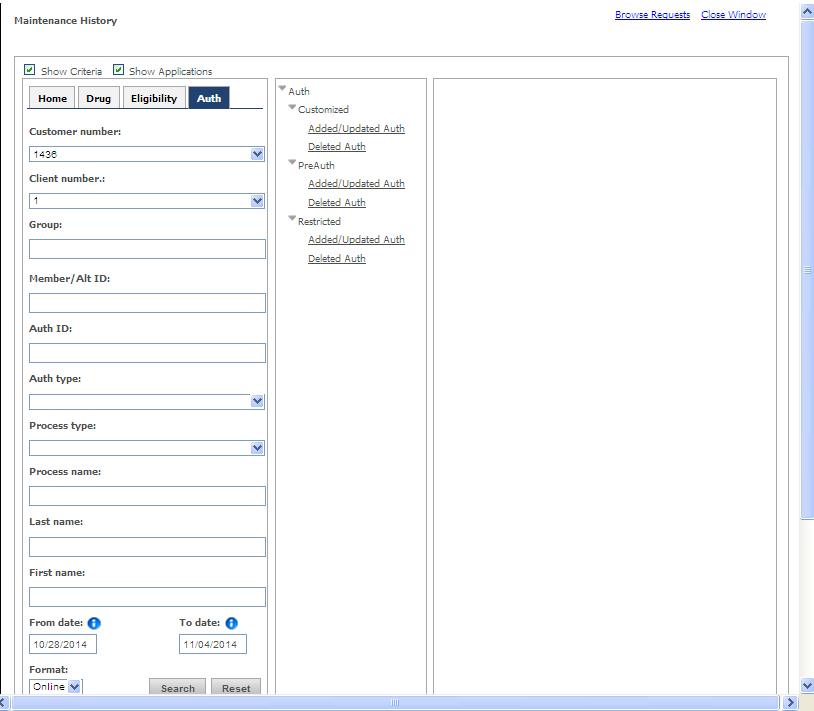
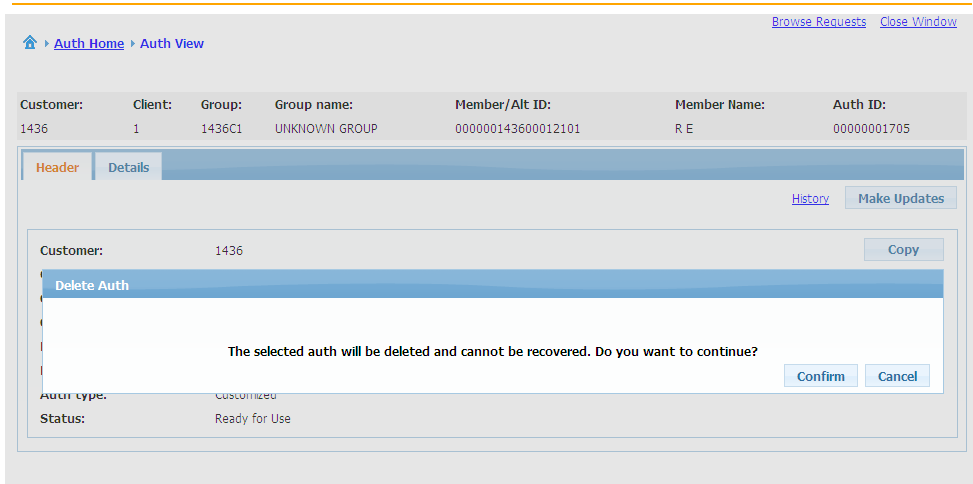
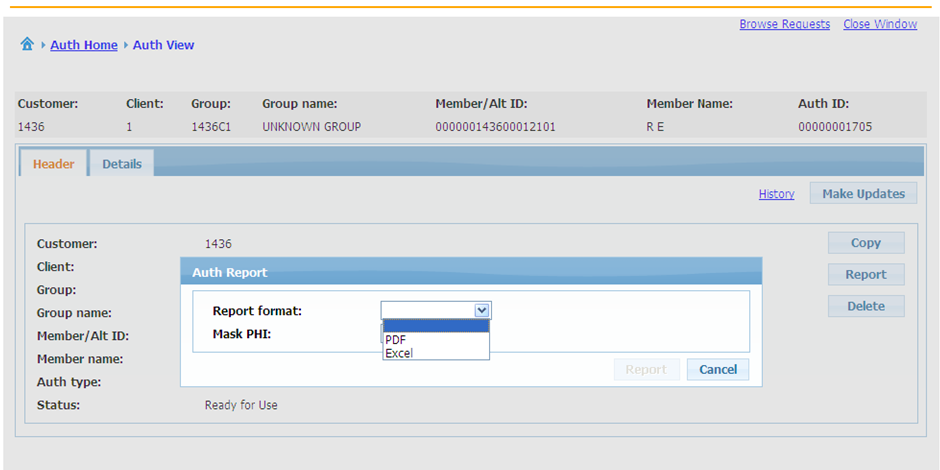
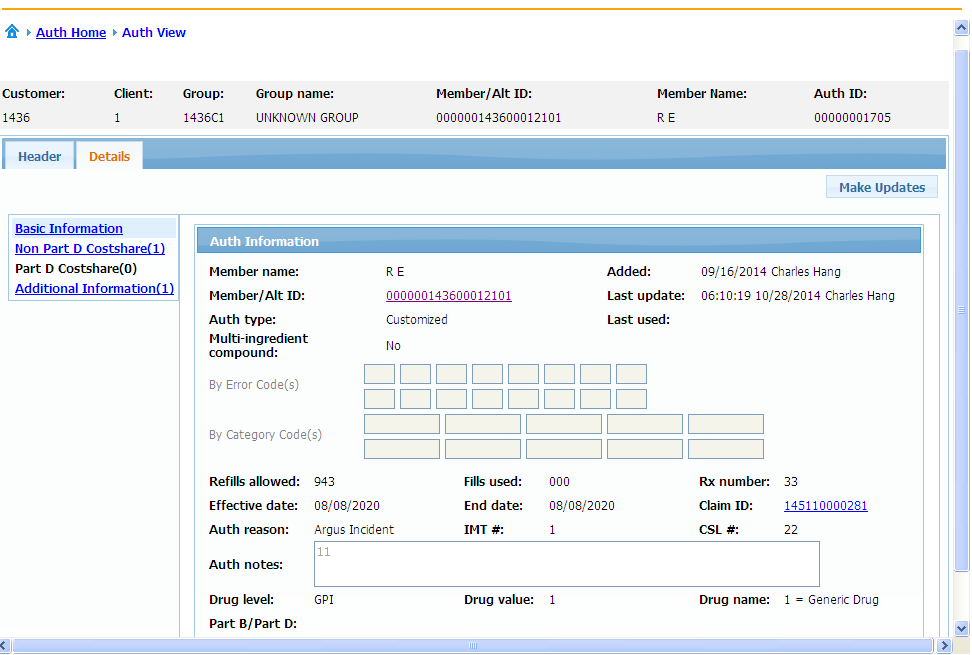
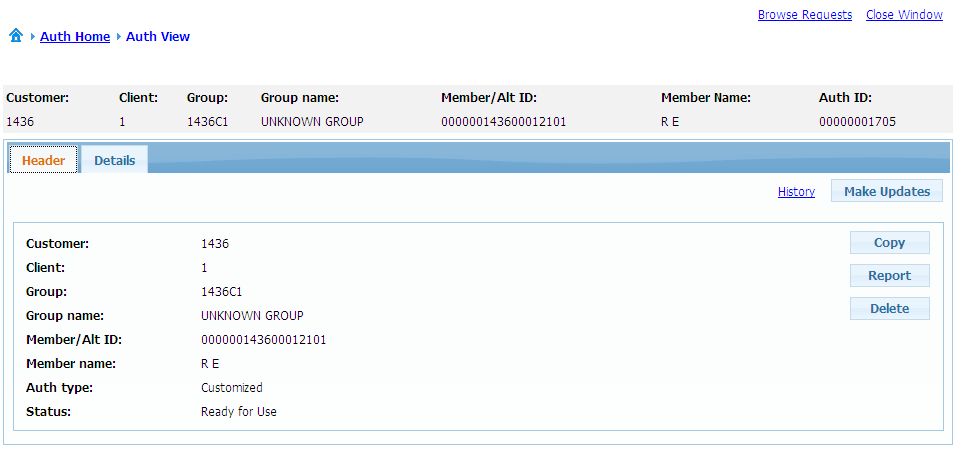
**2.6结合项目说明**

写测试用例的方法一般是I等价类划分法，比如Auth home页面的customerid，它的值的数据类型是DECIMAL(4, 0)，也就是说只能输入数字类型的，其他符号或字符都是不能输入的，并且只能输入4位数字，小数位保留0位，那这样复合这个规定的就是有效等价类，不符合的各种情况就是不同的无效等价类。II边界值划分法，根据域的输入输出数据类型，取刚刚大于或刚刚小于边界值的值作为测试的数据。III错误推断法，我会根据以往测试中常常出错的地方，来判断一下当前的测试会不会出现相同的错误。比如说search的条件和结果有时候往往不符合，或者说结果中只有部分符合，页码不够，条数不足，等等。

在Auth页面点击“New”按钮，会出现以下页面。



新建Preauth、Customized auth、Restricted auth这三种type的Auth，包括测试所需要的各种场景的数据。当新建的Auth被保存以后，页面会跳转到Auth view页面，可以进行Auth的删除、更新等的操作。到此所需要的数据已全部新建完毕。



**三、回归测试**

**3.1简介**

回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误。回归测试作为软件生命周期的一个组成部分，在整个软件测试过程中占有很大的工作量比重，软件开发的各个阶段都会进行多次回归测试。在渐进和快速迭代开发中，新版本的连续发布使回归测试进行的更加频繁，而在极端编程方法中，更是要求每天都进行若干次回归测试。因此，通过选择正确的回归测试策略来改进回归测试的效率和有效性是非常有意义的。

只要在程序内修改实现，都应该同样进行回归测试。具体的方法可以是：对修改过的代码重新运行现有的测试，确定更改是否破坏了在更改之前有效的任何事物，并且在必要的地方编写新测试。执行回归测试时，首要考虑的应该是覆盖范围足够大但不浪费时间。尽可能少花时间执行回归测试，但不减少在旧的、已经测试过的代码中检测新失败的可能性。

**3.2观念**

（1）回归测试是指重复以前的全部或部分的相同测试。

（2）新加入测试的模组，可能对其他模组产生副作用，故须进行某些程度的回归测试。

（3）回归测试的重心，以关键性模组为核心。

**3.3测试策略**

对于一个软件开发项目来说，项目的测试组在实施测试的过程中会将所开发的测试用例保存到“测试用例库”中，并对其进行维护和管理。当得到一个软件的基线版本时，用于基线版本测试的所有测试用例就形成了基线测试用例库。在需要进行回归测试的时候，就可以根据所选择的回归测试策略，从基线测试用例库中提取合适的测试用例组成回归测试包，通过运行回归测试包来实现回归测试。保存在基线测试用例库中的测试用例可能是自动测试脚本，也有可能是测试用例的手工实现过程。

回归测试需要时间、经费和人力来计划、实施和管理。为了在给定的预算和进度下，尽可能有效率和有效力地进行回归测试，需要对测试用例库进行维护并依据一定的策略选择相应的回归测试包。

3.3.1测试用例库的维护

为了最大限度地满足客户的需要和适应应用的要求，软件在其生命周期中会频繁地被修改和不断推出新的版本，修改后的或者新版本的软件会添加一些新的功能或者在软件功能上产生某些变化。随着软件的改变，软件的功能和应用接口以及软件的实现发生了演变，测试用例库中的一些测试用例可能会失去针对性和有效性，而另一些测试用例可能会变得过时，还有一些测试用例将完全不能运行。为了保证测试用例库中测试用例的有效性，必须对测试用例库进行维护。同时，被修改的或新增添的软件功能，仅仅靠重新运行以前的测试用例并不足以揭示其中的问题，有必要追加新的测试用例来测试这些新的功能或特征。因此，测试用例库的维护工作还应包括开发新测试用例，这些新的测试用例用来测试软件的新特征或者覆盖现有测试用例无法覆盖的软件功能或特征。

测试用例的维护是一个不间断的过程，通常可以将软件开发的基线作为基准，维护的主要内容包括下述几个方面。

1. 删除过时的测试用例

因为需求的改变等原因可能会使一个基线，测试用例不再适合被测试系统，这些测试用例就会过时。例如，某个变量的界限发生了改变，原来针对边界值的测试就无法完成对新边界测试。所以，在软件的每次修改后都应进行相应的过时测试用例的删除。

(2)改进不受控制的测试用例

随着软件项目的进展，测试用例库中的用例会不断增加，其中会出现一些对输入或运行状态十分敏感的测试用例。这些测试不容易重复且结果难以控制，会影响回归测试的效率，需要进行改进，使其达到可重复和可控制的要求。

(3)删除冗余的测试用例

如果存在两个或者更多个测试用例针对一组相同的输入和输出进行测试，那么这些测试用例是冗余的。冗余测试用例的存在降低了回归测试的效率。所以需要定期的整理测试用例库，并将冗余的用例删除掉。

(4)增添新的测试用例

如果某个程序段、构件或关键的接口在现有的测试中没有被测试，那么应该开发新测试用例重新对其进行测试。并将新开发的测试用例合并到基线测试包中。

通过对测试用例库的维护不仅改善了测试用例的可用性，而且也提高了测试库的可信性，同时还可以将一个基线测试用例库的效率和效用保持在一个较高的级别上。

3.3.2回归测试包的选择

在软件生命周期中，即使一个得到良好维护的测试用例库也可能变得相当大，这使每次回归测试都重新运行完整的测试包变得不切实际。一个完全的回归测试包括每个基线测试用例，时间和成本约束可能阻碍运行这样一个测试，有时测试组不得不选择一个缩减的回归测试包来完成回归测试。

回归测试的价值在于它是一个能够检测到回归错误的受控实验。当测试组选择缩减的回归测试时，有可能删除了将揭示回归错误的测试用例，消除了发现回归错误的机会。然而，如果采用了代码相依性分析等安全的缩减技术，就可以决定哪些测试用例可以被删除而不会让回归测试的意图遭到破坏。

选择回归测试策略应该兼顾效率和有效性两个方面。常用的选择回归测试的方式包括：

(1)再测试全部用例

选择基线测试用例库中的全部测试用例组成回归测试包，这是一种比较安全的方法，再测试全部用例具有最低的遗漏回归错误的风险，但测试成本最高。全部再测试几乎可以应用到任何情况下，基本上不需要进行分析和重新开发，但是，随着开发工作的进展，测试用例不断增多，重复原先所有的测试将带来很大的工作量，往往超出了我们的预算和进度。

(2)基于风险选择测试

可以基于一定的风险标准来从基线测试用例库中选择回归测试包。首先运行最重要的、关键的和可疑的测试，而跳过那些非关键的、优先级别低的或者高稳定的测试用例，这些用例即便可能测试到缺陷，这些缺陷的严重性也仅有三级或四级。一般而言，测试从主要特征到次要特征。

(3)基于操作剖面选择测试

如果基线测试用例库的测试用例是基于软件操作剖面开发的，测试用例的分布情况反映了系统的实际使用情况。回归测试所使用的测试用例个数可以由测试预算确定，回归测试可以优先选择那些针对最重要或最频繁使用功能的测试用例，释放和缓解最高级别的风险，有助于尽早发现那些对可靠性有最大影响的故障。这种方法可以在一个给定的预算下最有效的提高系统可靠性，但实施起来有一定的难度。

(4)再测试修改的部分

当测试者对修改的局部化有足够的信心时，可以通过相依性分析识别软件的修改情况并分析修改的影响，将回归测试局限于被改变的模块和它的接口上。通常，一个回归错误一定涉及一个新的、修改的或删除的代码段。在允许的条件下，回归测试尽可能覆盖受到影响的部分。

再测试全部用例的策略是最安全的策略，但已经运行过许多次的回归测试不太可能揭示新的错误，而且很多时候，由于时间、人员、设备和经费的原因，不允许选择再测试全部用例的回归测试策略，此时，可以选择适当的策略进行缩减的回归测试。

**3.4测试流程**

（1）识别出软件中被修改的部分；

（2）从原基线测试用例库“T”中，排除所有不再适用的测试用例，确定对新版本依然有效的测试用例，创建新的基线测试用例库“TN”；

（3）依据一定的策略从TN中选择测试用例测试被修改的软件；

（4）如果必要，生成新的测试用例集“T1”，用于测试TN无法充分测试的软件部分；

1. 用T1执行修改后的软件。

·第2和第3步测试验证修改是否破坏了现有的功能，第4和第5步测试验证修改工作本身。

**3.5集合项目说明**

当开发对软件进行了修改后，代码必然会发生变化后，也有可能引入新的BUG，比如说，我正在测试的auth web service，这一周的小项目已经完成，基本运行稳定，但是今天早上开发修改了代码，我就需要把相关case，重新测试一遍。