# 代码规范说明书

目录

[代码规范 1](#_Toc5904)

[一、命名规范 1](#_Toc30947)

[二、注释规范 1](#_Toc10825)

[三、排版规范 2](#_Toc15147)

[四、 变量结构规范 3](#_Toc23608)

[五、 可读性规范 5](#_Toc29989)

[六、 函数、过程规范 6](#_Toc4242)

[七、程序效率规范 9](#_Toc15455)

[八、测试规范 11](#_Toc27823)

## 一、命名规范

1、 项目名全部小写

2、 包名全部小写

3、 类名首字母大写，如果类名由多个单词组成，每个单词的首字母都要大写。

如：public class MyFirstClass{}

4、 变量名、方法名首字母小写，如果名称由多个单词组成，每个单词的首字母都要大写。

如：int index=0;

       public void toString(){}

5、 常量名全部大写

如：public static final String GAME\_COLOR=”RED”;

6、所有命名规则必须遵循以下规则：

1)、名称只能由字母、数字、下划线、$符号组成

2)、不能以数字开头

3)、名称不能使用JAVA中的关键字。

4)、坚决不允许出现中文及拼音命名。

## 

## 二、注释规范

1、   类注释

在每个类前面必须加上类注释，注释模板如下：

/\*\*

\* Copyright (C), 2006-2010, ChengDu Lovo info. Co., Ltd.

\* FileName: Test.java

\* 类的详细说明

\*

\* @author 类创建者姓名  
    \* @Date    创建日期

\* @version 1.00

\*/

2、   属性注释

在每个属性前面必须加上属性注释，注释模板如下：

/\*\* 提示信息 \*/

private String strMsg = null;

3、   方法注释

在每个方法前面必须加上方法注释，注释模板如下：

/\*\*

\* 类方法的详细使用说明

\*

\* @param 参数1 参数1的使用说明

\* @return 返回结果的说明

\* @throws 异常类型.错误代码 注明从此类方法中抛出异常的说明

\*/

4、   构造方法注释

在每个构造方法前面必须加上注释，注释模板如下：

/\*\*

\* 构造方法的详细使用说明

\*

\* @param 参数1 参数1的使用说明

\* @throws 异常类型.错误代码 注明从此类方法中抛出异常的说明

\*/

5、   方法内部注释

在方法内部使用单行或者多行注释，该注释根据实际情况添加。

如：//背景颜色

       Color bgColor = Color.RED

## 三、排版规范

1、相对独立的程序块之间必须加空行

下列情况应该使用一个空行：

两个方法之间

方法内的局部变量和方法的第一条语句之间

块注释或单行注释之前

一个方法内的两个逻辑段之间

1. 单行长度不超过120个字符，如超出则换行，且操作符应放在行首，新行要有相对缩进

3、多个参数之间，二元运算符左右，for语句表达式之间应用空格隔开表达式之间用空格隔 开

4、多个短语句不要写在一行中，也就是说一行只写一条语句

5、while、for、if、do、case、default、switch等语句独占一行，且while、for、do、if等语句的执行部分无论有多少句，都要用“{}”括起来

6、程序首行缩进4个空格数，不允许使用TAB缩进（使用空格可很好的适应不同操作系统的开发工具）

## 变量结构规范

1、去掉没必要的公共变量

说明：公共变量是增大模块间耦合的原因之一，故应减少没必要的公共变量以降低模块间的耦合度。

2、仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系

说明：在对变量声明的同时，应对其含义、作用及取值范围进行注释说明，同时若有必要还应说明与其它变量的关系。

3、明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等

说明：明确过程操作变量的关系后，将有利于程序的进一步优化、单元测试、系统联调以及代码维护等。

这种关系的说明可在注释或文档中描述。

示例：在源文件中，可按如下注释形式说明。

RELATION System\_Init Input\_Rec Print\_Rec Stat\_Score

Student Create Modify Access Access

Score Create Modify Access Access, Modify

注：RELATION 为操作关系；System\_Init、Input\_Rec、Print\_Rec、Stat\_Score 为四个不同的函数；Student、

Score 为两个全局变量；Create 表示创建，Modify 表示修改，Access 表示访问。

其中，函数 Input\_Rec、Stat\_Score 都可修改变量 Score，故此变量将引起函数间较大的耦合，并可能增加代码测试、维护的难度。

4、当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生

说明：对公共变量赋值时，若有必要应进行合法性检查，以提高代码的可靠性、稳定性。

5、防止局部变量与公共变量同名

说明：若使用了较好的命名规则，那么此问题可自动消除。

6、严禁使用未经初始化的变量作为右值

说明：特别是在 C/C++中引用未经赋值的指针，经常会引起系统崩溃。

7、构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的公共变量，防止多个不同模块或函数都可以修改、创建同一公共变量的现象

说明：降低公共变量耦合度。

8、使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量

说明：使用标准的数据类型，有利于程序的移植。

9、结构的功能要单一，是针对一种事务的抽象

说明：设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

10、要设计面面俱到、非常灵活的数据结构

说明：面面俱到、灵活的数据结构反而容易引起误解和操作困难。

11、不同结构间的关系不要过于复杂

说明：若两个结构间关系较复杂、密切，那么应合为一个结构。

typedef struct PERSON\_ONE\_STRU

{

unsigned char name[8];

unsigned char addr[40];

unsigned char sex;

unsigned char city[15];

} PERSON\_ONE;

typedef struct PERSON\_TWO\_STRU

{

unsigned char name[8];

unsigned char age;

unsigned char tel;

} PERSON\_TWO;

由于两个结构都是描述同一事物的，那么不如合成一个结构。

typedef struct PERSON\_STRU

{

unsigned char name[8];

unsigned char age;

unsigned char sex;

unsigned char addr[40];

unsigned char city[15];

unsigned char tel;

} PERSON;

12、结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构，以减少原结构中元素的个数

13、仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象

说明：合理排列结构中元素顺序，可节省空间并增加可理解性。

示例：如下结构中的位域排列，将占较大空间，可读性也稍差。

typedef struct EXAMPLE\_STRU

{

unsigned int valid: 1;

PERSON person;

unsigned int set\_flg: 1;

} EXAMPLE;

若改成如下形式，不仅可节省 1 字节空间，可读性也变好了。

typedef struct EXAMPLE\_STRU

{

unsigned int valid: 1;

unsigned int set\_flg: 1;

PERSON person ;

} EXAMPLE;

14、结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地（如预留一些空间等）

15、留心具体语言及编译器处理不同数据类型的原则及有关细节

说明：如在 C 语言中，static 局部变量将在内存“数据区”中生成，而非 static 局部变量将在“堆栈”中生成。

这些细节对程序质量的保证非常重要。

16、编程时，要注意数据类型的强制转换

说明：当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而这些细节若考虑不周，就很有可能留下隐患。

17、对编译系统默认的数据类型转换，也要有充分的认识

示例：如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

char chr;

unsigned short int exam;

chr = -1;

exam = chr; // 编译器不产生告警，此时 exam 为 0xFFFF。

18、尽量减少没有必要的数据类型默认转换与强制转换

19、合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换

20、对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其命名方式在

同一产品中的统一

说明：使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序清晰、简洁。

## 可读性规范

1. 尽量使方法简短。上下文很重要，5-20行为最佳。（始终能在一个屏幕里放得下）  
   2.不要把一个变量用于多个服务。（一个变量应该始终只为一个目的服务）  
   3.使变量能顾名思义。（ 只要看一眼就能知道是干嘛的）  
   4.把变量定义在靠近使用的地方。同1的上下文的原因。  
   5.拒绝神秘数字 a < 4386 不如 a < MAX\_INPUT\_LENGTH 。在使用一个数值前赋给变量。  
   6.不要逆常规。如驼峰表示法，常量定义等的书写方式。  
   7.警惕过早优化。你第一应该关心的事情是写出易于理解的代码。如果你觉得有些地方很慢，不要盲目的重写代码，你应先找到慢的证据。不要傻乎乎的去解决根本不存在的问题 。  
   8.不要过度沉迷于技巧，比如 设计模式 一个80行就能写完的脚本，你最终使用了10个类，15个接口，外加一大堆范式和标记符。 97%的代码不做任何事情。  
   9.避免出现盗梦空间式的嵌套。

## 函数、过程规范

1、对所调用函数的错误返回码要仔细、全面地处理

2、防止将函数的参数作为工作变量

说明：将函数的参数作为工作变量，有可能错误地改变参数内容，所以很危险。对必须改变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

示例：下函数的实现不太好。

void sum\_data( unsigned int num, int \*data, int \*sum )

{

unsigned int count;

\*sum = 0;

for (count = 0; count < num; count++)

{

\*sum += data[count]; // sum 成了工作变量，不太好。

}

}

若改为如下，则更好些。

void sum\_data( unsigned int num, int \*data, int \*sum )

{

unsigned int count ;

int sum\_temp;

sum\_temp = 0;

for (count = 0; count < num; count ++)

{

sum\_temp += data[count];

}

\*sum = sum\_temp;

}

3、函数的规模尽量限制在 200 行以内

说明：不包括注释和空格行。

4、一个函数仅完成一件功能

5、为简单功能编写函数

说明：虽然为仅用一两行就可完成的功能去编函数好象没有必要，但用函数可使功能明确化，增加程序可

读性，亦可方便维护、测试。

示例：如下语句的功能不很明显。

value = ( a > b ) ? a : b ;

改为如下就很清晰了。

int max (int a, int b)

{

return ((a > b) ? a : b);

}

value = max (a, b);

或改为如下。

#define MAX (a, b) (((a) > (b)) ? (a) : (b))

value = MAX (a, b);

6、不要设计多用途面面俱到的函数

说明：多功能集于一身的函数，很可能使函数的理解、测试、维护等变得困难

7、函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出

8、尽量不要编写依赖于其他函数内部实现的函数

9、避免设计多参数函数，不使用的参数从接口中去掉

说明：目的减少函数间接口的复杂度。

10、非调度函数应减少或防止控制参数，尽量只使用数据参数

说明：本建议目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令，来启动相应的功能实体（即函数或过程），而本身并不完成具体功能。控制参数是指改变函数功能行为的参数，即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合，很可能使函数间的耦合度增大，并使函数的功能不唯一。

示例：如下函数构造不太合理。

int add\_sub( int a, int b, unsigned char add\_sub\_flg )

{

if (add\_sub\_flg == INTEGER\_ADD)

{

return (a + b);

}

else

{

return (a b);

}

}

不如分为如下两个函数清晰。

int add( int a, int b )

{

return (a + b);

}

int sub( int a, int b )

{

return (a b);

}

11、检查函数所有参数输入的有效性

12、检查函数所有非参数输入的有效性，如数据文件、公共变量等

说明：函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。

函数在使用输入之前，应进行必要的检查。

13、函数名应准确描述函数的功能，使用动宾词组为执行某操作的函数命名。如果是 OOP 方法，可以只有动词（名词是对象本身），避免使用无意义或含义不清的动词为函数命名

说明：避免用含义不清的动词如 process、handle 等为函数命名，因为这些动词并没有说明要具体做什么。

14、函数的返回值要清楚、明了，让使用者不容易忽视错误情况

说明：函数的每种出错返回值的意义要清晰、明了、准确，防止使用者误用、理解错误或忽视错误返回码。

15、除非必要，最好不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回

16、在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换

说明：因为数据类型转换或多或少存在危险。

17、防止把没有关联的语句放到一个函数中

说明：防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。

在编程时，经常遇到在不同函数中使用相同的代码，许多开发人员都愿把这些代码提出来，并构成一个新

函数。若这些代码关联较大并且是完成一个功能的，那么这种构造是合理的，否则这种构造将产生随机内聚的函数。

示例：如下函数就是一种随机内聚。

void Init\_Var( void )

{

Rect.length = 0;

Rect.width = 0; /\* 初始化矩形的长与宽 \*/

Point.x = 10;

Point.y = 10; /\* 初始化“点”的坐标 \*/

}

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。

应如下分为两个函数：

void Init\_Rect( void )

{

Rect.length = 0;

Rect.width = 0; /\* 初始化矩形的长与宽 \*/

}

void Init\_Point( void )

{

Point.x = 10;

Point.y = 10; /\* 初始化“点”的坐标 \*/

}

18、如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题

说明：若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成一个新的函数。

功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在

说明：模块中函数划分的过多，一般会使函数间的接口变得复杂。所以过小的函数，特别是扇入很低的或功能不明确的函数，不值得单独存在。

19、设计高扇入、合理扇出（小于 7 ）的函数

说明：扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。

扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，如总是 1，表明函数的调用层次可能过多，这样不利程序阅读和函数结构的分析，并且程序运行时会对系统资源如堆栈空间等造成压力。函数较合理的扇出（调度函数除外）通常是 3-5。扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。

扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

20、 减少函数本身或函数间的递归调用

说明：递归调用特别是函数间的递归调用（如 A->B->C->A），影响程序的可理解性；递归调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便，应减少没必要的递归调用。

21、当一个过程（函数）中对较长变量（一般是结构的成员）有较多引用时，可以用一个意义相当的宏代替

## 七、程序效率规范

1、编程时要经常注意代码的效率  
说明：代码效率分为全局效率、局部效率、时间效率及空间效率。全局效率是站在整个系统的角度上的系统效率；局部效率是站在模块或函数角度上的效率；时间效率是程序处理输入任务所需的时间长短；空间效率是程序所需内存空间，如机器代码空间大小、数据空间大小、栈空间大小等。  
2、在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率  
说明：不能一味地追求代码效率，而对软件的正确性、稳定性、可读性及可测性造成影响。  
3、局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响  
4、通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率  
说明：这种方式是解决软件空间效率的根本办法。  
示例：如下记录学生学习成绩的结构不合理。

typedef unsigned char BYTE; typedef unsigned short WORD; typedef struct STUDENT\_SCORE\_STRU

BYTE name[8];

BYTE age;

BYTE sex;

BYTE class;

BYTE subject;

float score;

} STUDENT\_SCORE;

因为每位学生都有多科学习成绩，故如上结构将占用较大空间。应如下改进（分为两个结构），总的存贮空间将变小，操作也变得更方便。

typedef struct STUDENT\_STRU

{

BYTE name[8];

BYTE age;

BYTE sex;

BYTE class;

} STUDENT; typedef struct STUDENT\_SCORE\_STRU

{

WORD student\_index;

BYTE subject;

float score;

} STUDENT\_SCORE;

5、循环体内工作量最小化  
说明：应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。  
6、对模块中函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高程序效率  
说明：软件系统的效率主要与算法、处理任务方式、系统功能及函数结构有很大关系，仅在代码上下功夫一般不能解决根本问题。  
7、不应花过多的时间拼命地提高调用不很频繁的函数代码效率  
8、要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数  
说明：只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方式。嵌入汇编可提高时间及空间效率，但也存在一定风险。  
9、：在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量，来提高空间效率  
说明：这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。  
10、在多重循环中，应将最忙的循环放在最内层  
说明：减少 CPU 切入循环层的次数。  
示例：如下代码效率不高。

for (row = 0; row < 100; row++)

{

for (col = 0; col < 5; col++)

{

sum += a[row][col];

}

}

可以改为如下方式，以提高效率。

for (col = 0; col < 5; col++)

{

for (row = 0; row < 100; row++)

{

sum += a[row][col];

}

}

11、尽量减少循环嵌套层次  
12、避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中  
说明：目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不可以。  
13、尽量用乘法或其它方法代替除法，特别是浮点运算中的除法  
说明：浮点运算除法要占用较多 CPU 资源。  
示例：如下表达式运算可能要占较多 CPU 资源。

#define PAI 3.1416

radius = circle\_length / (2 \* PAI);

应如下把浮点除法改为浮点乘法。

#define PAI\_RECIPROCAL (1 / 3.1416 ) // 编译器编译时，将生成具体浮点数

radius = circle\_length \* PAI\_RECIPROCAL / 2;

## 八、测试规范

主要从以下几个方面入手

角色的确定，进入准则，输入项，活动过程，输出项，评审与评估，退出准则，度量

1、角色的确定

测试组长：业务专家，负责项目的管理，测试计划的制定，项目文档的审查，测试用例的设计和审查，任务的安排，与项目经理和开发组长沟通等等

实验室管理人员：设置配置和维护实验室环境，主要是服务器和网络环境等等

资深测试工程师：负责产品设计规格说书的审查，测试用例的设计和技术难题的解决，主要参与数据库，系统性能与安全性能等技术难度较高的测试

自动化测试功能师：负责测试工具的开发和测试脚本的开发等

初级测试工程师：执行测试用例和相关测试任务，测试功能测试用例的设计与执行

2、进入准则

全程测试！ 无时无刻不在测试，注意测试的节点进度

3、输入项

即需求资料，这些是判断软件是否符合需求的依据和标准；

软件项目项目计划 软件项目计划是一个综合的项目信息载体，用来收集管理项目时所需要的所有信息

软件需求文档：描述软件需求的文档，如是需求文档MRD，用户故事user story，用例，用户行为和产品特性列表，以及利用case /UML工具建模生成的文档

软件架构设计文档：系统架构设计文档主要描述系统整体结构方案、软件子系统划分，子系统间的接口和错误处理机制等

软件详细设计文档：详细设计文档主要描述模块或组件、类、对象、等中的元素、如数据字典，对象属性、类的方法、和全局变量等等

软件程序单元：包括所有已完成的程序单元源代码、数据库脚本、系统配置文件等等

软件配置或集成计划 ：软件工作版本的定义，工作版本的内容，集成的策略以及实施的先后顺序；

软件工作版本：按照集成计划完创建完成的各个集成工作版本！

4、活动过程

1）制定测试计划

角色：测试设计人员，组长，资深测试工程师

具体做法如下：

制定测试计划吗----收集和组织测试计划信息，并且创建测试计划

确定测试需求----根据需求规格说明书，质量计划等收集和整理测试需求信息，确定质量需求和测试目标

制定测试策略—针对测试需求，定义测试阶段，测试类型，测试方法，测试风险回避措施以及所需要的测试工具

建立测试通过标准—根据项目实际情况。为每一个层次的测试或每一个阶段简历通过准则

确定资源和进度—确定测试所需的软件硬件资源，人力资源以测试进度

评审测试计划----修正评审中发现的问题，达成一致意见，最终签发

2)测试设计

角色：测试设计人员，

活动描述：设计测试的目的就是如何有效的完成测试需要所确定的测试任务，为每一个测试需求确定要执行的测试任务，测试脚本或用例集，并且明确测试执行过程

设计测试用例：

》为每一个测试，确定其所需要的测试用例

》为每一个测试用例，确定其输入以及预期结果

》根据界面原型为每一个测试用例定义详细的测试步骤

》确定测试用例的测试环境配置、前置条件和后置条件

》为测试用例准备输入数据

》编写测试用例文档

》对测试用例进行评审

3）开发测试工具和脚本

角色：自动化测试工程师，测试工程师以及程序员

活动描述：

实施测试的目的是创建可重用的测试脚本，并且实施测试驱动程序和桩程序

》根据测试过程，创建、开发测试脚本，并且调试测试脚本

》根据设计编写测试需要的测试驱动程序和桩程序

4)执行单元测试

角色：程序员为主，测试员为辅

活动描述：

执行单元测试的目的就是验证单元内部结构以及单元实现的功能

具体做法如下：

》按照测试过程，手工执行单元测试或运行测试脚本自动执行测试

》详细记录单元测试结果，并且评估测试结果，直至达到测试覆盖率

》回归测试—对修改后的单元执行回归测试

5）执行集成测试

角色：

程序员和测试工程师

活动描述：执行集成测试的目的是验证单元之间的接口是否一致，是否磕炮等，现在采用的“”“持续集成的模式”，单元测试和集成测试一般同时进行

》执行集成测试-----按照测试过程，手工执行集成测试或者运行自动化脚本测试执行测试

》详细记录集成测试的结果，并将测试结果提交给相关测试人员

6）执行系统测试

角色：资深测试工程师，测试实验室管理员

活动描述：执行系统测试的目的是确认集成后的软件系统不进满足功能性需求，还满足非功能性需求，如性能，安全性，兼容性等

》执行系统测试----手工执行或者运行测试脚本，自动执行系统测试

》详细记录系统测试结果，并对测试结果进行分析，提交测试结果和分析报告给先关人员

7）评估测试

角色：测试人员以及相关技术人员

活动描述：评估测试的目的是对每次测试结果进行分析和评估，提交测试分析报告，并且根据评估结果，决定是否需要对测试计划进行修改，对下一次测试活动做出调整

》分析测试结果----有测试人员对每一次测试结果进行分析，并提出变更请求或其他处理意见

》评估阶段测试状态和产品质量状态；如评估测试覆盖率，测试结果，对缺陷进行分析等等

5、输出项

1）软件测试计划

包括：项目范围内的质量要求，测试目标，测试需求的有关信息；此外，测试计划还确定所需的资源、测试环境、进度安排等等

2）软件测试用例 ：测试用例是为特定目标开发的测试输入、执行条件、和预期结果的集合

3）测试缺陷报告

测试结果记录测试期间测试用例的执行情况，记录测试发现的缺陷，并且用来对缺陷进行跟踪

4）测试分析报告

测试分析报告是对每一个阶段测试（单元测试，集成测试，系统测试）的测试结果进行的分析评估

6、评估和评审

软件测试计划的评审

软件测试用例的评审

软件分析报告的评审

质量保证评审