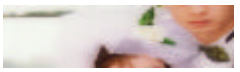
密　　级：普通
文件编号：NO.1
文件类别：测试管理体系文件
发 放 号：1001

应用软件

单元测试规范

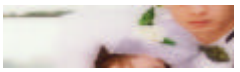
版本：1.0

北京梅梅出品有限公司



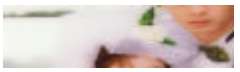
版本说明

日期	版本号	发布说明	作者	批准人	
				签字	岗位



目录

1 引言 3
1.1 编写目的 3
1.2 背景 3
1.3 定义 3
1.4 参考文档 3
2 单元测试 4
2.1 单元的定义 4
2.2 角色工作体系 4
2.3 单元测试规程 4
2.4 单元测试工具 5
2.5 测试的目录结构 5
2.6 测试代码的书写规范 6
2.7 测试单元的文件组成及命名规范 6
2.8 单元测试的实施 6
3 测试结果提交和验收 8
3.1 单元测试工作产品提交 8
3.2 单元测试工作产品验收规范 9
附录一：代码审查单 10
附录二：单元测试 Bug 清单 13
附录三：驱动模块（类）模板 14
附录四：单元测试实例介绍 17



1 引言

1.1 编写目的

1.1.1 编写目的

本文档规定了应用软件系统和部分系统平台模块的单元测试方法和步骤、测试用例的设计方法、测试代码的书写规范、流程以及单元测试的产品提交和验收规范，目的在于控制单元测试的质量，加强项目的质量管理，从而提高整个产品的质量。

1.1.2 适用范围

主要是应用软件的单元测试、部分系统平台软件模块测试。

1.1.3 预期读者

本文档的预期读者为项目的项目经理、产品经理、系统软件主研人员、应用软件主研人员、高级测试人员等。

1.2 背景

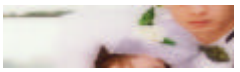
XXXXXX 系统软件平台是项目的重要组成部分，主要是依托 GUI 子系统、分析子系统和数据采集子系统的硬件环境，共同为高层的应用软件提供必要的软、硬件功能支持，并为应用软件开发人员提供必要的开发环境和测试环境。本规范的提出和制订旨在为软件单元测试提供依据和支持。

1.3 定义

被测模块：需要进行模块级测试的应用软件系统的一个单元或模块，也称被测单元
测试单元：用于对被测模块进行单元级测试，由源代码、测试脚本和输入数据等构成的程序单元

1.4 参考文档

[1] CppUnit Documentation
[2] gprof homepage
[3] gcov homepage
[4] 应用软件编写规范
[5] DemoUnit 测试单元
[6] 单元测试培训材料
[7] 单元测试总结报告



2 单元测试

2.1 单元的定义

对于结构化的编程语言，程序单元指程序中定义的函数或子程序。单元测试是指对函数或子程序所进行的测试。

对于面向对象的编程语言，程序单元指特定的一个具体的类或相关的多个类。单元测试主要是指对类方法的测试。

2.2 角色工作体系

角色	职责
测试主管	审查单元测试过程，对测试结果进行评估。根据单元测试发现的缺陷提出变更申请。
测试工程师	对单元代码进行检查，设计单元测试用例，加载运行测试用例，记录和分析测试结果，填写单元测试 Bug 清单。
开发工程师	设计测试需要的驱动程序和桩模块，以及辅助测试工具的开发。
配置管理员	管理测试需要的资源，包括软硬件环境，版本管理和 Bug 管理。

2.3 单元测试规程

包括静态的代码审查和动态测试两个阶段。

代码审查是按照《代码审查单》中的条项对单元模块进行逐项检查，并填写《单元测试 Bug 清单》。《代码审查单》的格式见附录一，《单元测试 Bug 清单》见附录二。

动态测试阶段首先编写驱动模块（或主类）和桩模块后，在驱动模块和桩模块中设计相应的测试用例，对所有的测试用例进行统一编号，在源代码中进行注释标识。测试用例应该覆盖单元模块的所有功能项，如果单元模块有性能、余量等其它测试特性要求，则必须设计相应的测试用例测试这些特性，编制完测试用例后，把测试用例提交给配置管理员或测试主管进行审查，审查没有通过则根据审查意见进行修改，直到审查通过后测试人员加载测试用例，编译运行得到测试结果，比对测试结果，如果发现错误或 Bug 则需要填写《单元测试 Bug 清单》并提交给测试经理和配置管理人员。

在进行功能测试时，可以利用其它测试工具进行内存溢出分析、代码覆盖率分析、代码性能测试等。

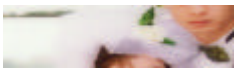
2.3.1 代码审查

要求：根据《代码审查单》中的要求，对被测试单元进行逐项检查，检查后在对应的条项后进行标记，发现问题后，填写《代码单元测试 Bug 清单》并提交。

2.3.2 测试用例

测试用例是测试数据及与之相关的测试规程的一个特定的集合，它是为验证被测试程序（为测试路径或验证是否符合特定需求）而产生的。

测试用例设计用于白盒测试和黑盒测试。



白盒测试进入的前提条件是在测试人员已经对被测试对象有了一定的了解，基本上明确了被测试软件的逻辑结构。过程是通过针对程序逻辑结构设计和加载测试用例，驱动程序执行，检查在不同点程序的状态，以确定实际的状态是否与预期的状态一致。

白盒测试主要是对被测试对象进行如下测试项目：

1、 对程序模块的所有独立的执行路径至少覆盖一次；

2、 对所有的逻辑判定，真假两种情况都至少覆盖一次；

3、 在循环的边界和运行界限内执行循环体；

4、 测试内部数据结构的有效性等。

白盒测试达到的目标：语句覆盖率达到 100%，分支覆盖率达到 100%，覆盖程序中主要的路径，主要路径是指完成需求和设计功能的代码所在的路径和程序异常处理执行到的路径。

黑盒测试是要首先了解软件产品具备的功能和性能等需求，再根据需求设计一批测试用例以验证程序内部活动是否符合设计要求的活动。

黑盒测试主要是对被测试对象进行如下测试项目：

1、 测试程序单元的功能是否实现；

2、 测试程序单元性能是否满足要求（可选）；

3、 可选的其它测试特性，如边界、余量、安全性、可靠性、强度测试、人机交互界面测试等。

黑盒测试达到的目标：程序单元正确地实现了需求和设计上要求的功能，满足性能要求，同时程序单元要有可靠性和安全性。

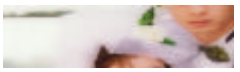
2.4 单元测试工具

项目规定使用以下测试工具实现应用软件系统单元测试和子系统集成测试，以及部分系统平台软件模块的相关测试。

- CppUnit：正确性测试和功能测试
- ccmalloc：动态内存访问检查
- gcov：代码覆盖率分析
- gprof：代码性能分析

2.5 测试的目录结构

建议将模块单元的测试代码组织在一个单独的目录中，作为模块单元源代码目录的一个子目录，取名为 TestDemo。在测试代码目录下分布创建 5 个子目录分别对应 PC Linux、PXA250 评估板、IXP425 评估板、PXA255 目标板、IXP425 目标板的测试目录，用于构建、执行单元测试、管理测试日志和测试报告。



2.6 测试代码的书写规范

其规范见附录三。

2.7 测试单元的文件组成及命名规范

每个测试单元由测试代码文件、程序主函数文件和编译运行脚本文件组成，单元测试完成之后还生成一系列测试报告，这些测试报告将与模块单元一起提交。

为了便于管理，对组成测试单元的各个文件及测试生成的测试结果和测试报告文件的命名都从被测类/模块派生而来。假定被测类为 DemoClass，测试单元包含如下文件及其所处目录位置如下所述：

1) 测试单元文件

TestDemo/DemoClassTest.h：测试类头文件

TestDemo/DemoClassTest.cpp：测试类实现文件

TestDemo/DemoUnitMain.cpp：测试类主函数

TestDemo/$(运行平台)/Makefile：用于特定运行平台的 makefile 文件

TestDemo/$(运行平台)/DemoTestDemo：为特定运行平台生成的可执行程序

其中运行平台为：PC Linux、PXA250 评估板、PXA255 目标板、IXP425 评估板、IXP425 目标板 5 种。

2) 测试结果文件

TestDemo/$(运行平台)/DemoUnit-00.log：采用-00 编译的正确性测试结果文件

TestDemo/$(运行平台)/DemoUnit-02.log：采用-02 编译的正确性测试结果文件

TestDemo/$(运行平台)/DemoUnit-03.log：采用-03 编译的正确性测试结果文件

TestDemo/$(运行平台)/DemoUnit.ccmalloc：内存检查结果文件

TestDemo/$(运行平台)/DemoClass.gcov：DemoClass.cpp 的代码覆盖率结果文件

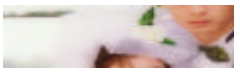
TestDemo/$(运行平台)/DemoUnit.gprof：DemoUnit 被测单元的代码性能分析结果文件

其中运行平台为：PC Linux、PXA250 评估板、PXA255 目标板、IXP425 评估板、IXP425 目标板

2.8 单元测试的实施

按照单元测试规程进行实施，进行代码审查和动态测试。

1) 单元测试或集成测试涉及的源程序三种：被测类/被测单元、已通过的类/桩模块、测试单元。只需对被测类进行测试设计、进行代码覆盖率分析和代码性能分析，用多种优化编译选项进行编译和测试；
2) 不需为已通过的类/桩模块进行测试设计，这些模块单元和测试单元本身都进行代码不需要使用 ccmalloc、gcov 和 gprof 等工具要求的编译选项和编译优化选项进行编译，也不需要为其生成.gcov 代码覆盖率报告。
3) 对于各种运行平台下，都需要使用-00，-02，-03 三种编译优化选项对测试单元进行编译，并运行一个测试单元中的所有测试用例，生成测试报告



2.8.1 单元模块正确性测试

进行单元正确性测试的过程是将被测单元源程序、测试单元源程序和测试主函数程序放到一起编译产生可执行程序，并在目标平台上运行可执行程序，即可获得测试结果报告。对应上述的 DemoClass 被测类的正确性测试过程的命令序列为：

```
$(CC) $(OPT) -c DemoClass.cpp              ;编译被测类
$(CC) -c DemoClassTest.cpp
$(CC) -c DemoUnitMain.cpp
$(CC) -o DemoTestDemo DemoClass.o DemoClassTest.o DemoUnitMain.o -lstdc++ -lcppunit
./DemoTestDemo                             ;运行测试
./DemoTestDemo DemoUnit$(OPT).log          ;生成单元测试结果文件，该文件随模块一起提交
```

其中，变量 CC 为 C/C++编译器，如 gcc/g++；$(OPT)为编译优化选项。

项目要求每个被测模块在用-O0，-O2 和-O3 三种编译选项进行编译，并分别进行正确性测试。

2.8.2 单元内存溢出检查

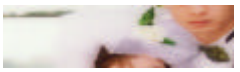
项目要求用 ccmalloc 内存检查工具对被测单元进行内存溢出检查，测试过程与正确性测试相似，只是要求被测单元代码的编译和最后的连接命令前添加 ccmalloc 命令，如下命令序列所示：

```
ccmalloc $(CC) $(OPT) -c DemoClass.cpp
$(CC) -c DemoClassTest.cpp
$(CC) -c DemoUnitMain.cpp
ccmalloc $(CC)  -o DemoTestDemo DemoClass.o DemoClassTest.o DemoUnitMain.o  -lstdc++ -lcppunit
./DemoTestDemo                             ;运行测试，产生内存检查结果显示于屏幕
./DemoTestDemo 2> DemoUnit.ccmalloc  ; 运行测试，产生内存检查结果文件用于提交
```

2.8.3 测试代码覆盖率分析

项目要求用 gcov 工具对测试单元的代码覆盖率进行分析，测试单元的代码覆盖率分析的命令序列如下所示：

```
$(CC) $(OPT) -c -g -fprofile-arcs -ftest-coverage DemoClass.cpp  -fprofile-arcs
  ;对被测代码使用-g -ftest-coverage 等编译选项
$(CC) -c DemoClassTest.cpp
$(CC) -c DemoUnitMain.cpp
$(CC) -o DemoTestDemo DemoClass.o DemoClassTest.o DemoUnitMain.o -lstdc++ -lcppunit
./DemoTestDemo                                  ;运行测试
gcov DemoClass.cpp > DemoClass.gcov.sum         ;对每个被测源程序生成 2 个覆盖率结果文件
                                                ; DemoClasscpp.gcov 和 DemoClass.gcov.sum
                                                ;前者包含源代码每条语句的执行计数，
                                                ;后者包含一个该文件覆盖率统计
 cat DemoClass.gcov.sum DemoClass.cpp > DemoClass.gcov    ;合并以上两个代码覆盖率文件，
```



;最后提交合并后的文件

2.8.4 模块单元代码性能分析

项目还要求用 gcov 工具对测试单元的代码性能进行分析，测试单元的代码性能分析的命令序列如下所示：

```
$(CC) $(OPT) -c -g -pg DemoClass.cpp          ;对被测类使用-g -pg 等编译选项
$(CC) -c DemoClassTest.cpp
$(CC) -c DemoUnitMain.cpp
$(CC) -pg -o DemoTestDemo DemoClass.o DemoClassTest.o DemoUnitMain.o -lstdc++ -lcppunit
./DemoTestDemo                                ;运行测试
gprof -pg DemoTestDemo >DemoUnit.prof         ;产生性能分析结果文件
```

3 测试结果提交和验收

3.1 单元测试工作产品提交

项目要求随模块提交 2.8 列出的 5 种测试单元文件和 6 种测试结果和测试报告文件，而每增加一种被测类，提交时要求增加相应的测试类文件和代码覆盖率报告文件。

3.1.1 提交的测试产品

1 对于每个被测类的测试文档产品

- 测试类头.h 文件
- 测试类实现.cpp 文件
- PC Linux 平台和 2 个 XScale 平台(2 个 PXA25X 平台或 2 种 IXP425 平台)下的代码覆盖率.gcov 文件

2 对于每个测试单元的测试文档产品

- 测试类主函数.cpp 文件

3 对于每种运行平台的测试文档产品

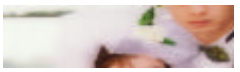
对于每个测试单元需要提在 PC Linux 平台和 2 个 XScale 平台（2 个 PXA25X 平台或 2 种 IXP425 平台）下的以下文档

- Makefile 文件
- 内存检查结果.ccmalloc 文件
- 代码覆盖率分析.gcov 文件
- 代码性能分析.gprof 文件
- 利用-O0，-O2，-O3 三种编译优化选项编译被测代码时产生正确性测试结果.log 文件

4 单元测试总结报告.report

TestDemo/DemoUnit.report：总结单元测试情况，需要手工书写。内容包括 4 个部分：

- 被测类名：列出所有被测类的类名
- 测试用例：按被测类列出所有测试用例及其描述信息,主要是用例源程序代码和相应的注释信息。
- 正确性测试报告：列出每种运行平台下测试单元运行的测试结果。从具有最高编译选项并且通



过了全部测试用例的测试报告中拷贝

- 代码覆盖率测试结果：列出测试单元在任意平台下运行时，被测类的代码覆盖率信息。从相应被测类的.gcov 文件中拷贝。

  一个 Demo 单元测试总结报告请参考 DemoUnit.report[9]。

3.1.2 测试产品提交方式

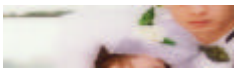
单元编码/测试人员应该在所有测试项目完成之后，删除所有无关的临时文件，仅留下需要提交的项目，然后将 TestDemo 目录作为一个整体保留其目录结构进行提交。最后手工完成一个文本格式的单元测试总结报告。

3.2 单元测试工作产品验收规范

项目的模块单元提交时，要对-O0、-O2 和-O3 三种编译优化的正确性测试报告.log 文件、每个被测类/被测源文件的代码覆盖率结果.gcov 文件和内存检查结果.ccmalloc 文件。

通过的准则如下：

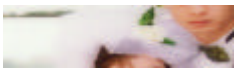
1) 正确性测试结果文件：在所有运行平台下，至少在一种编译优化选项下通过了全部的测试用例，保证测试用例覆盖了单元模块中的所有功能点；
2) 其它测试特性结果文件：在所有运行平台下，测试覆盖该模块所要求的其它测试特性并测试通过；
3) 内存检查结果文件：在所有运行平台下，运行所有测试用例之后未发生内存泄漏；
4) 代码覆盖率文件：在所有运行平台下，每个被测类/被测文件的可执行语句的代码覆盖率达到 100%；
4) 每一个单元测试 Bug 清单都处于一个明确的状态，不能改正的必须给出详细的解释说明；
5) 单元测试工作产品的验收采用同级评审的方法，由评审组决定测试是否通过，来保证单元测试的质量和软件产品的质量。



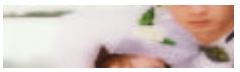
附录一：代码审查单

代码审查单

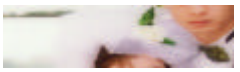
检查大项	检查小项	是否
编程风格检查	按照代码编写规范，该缩进的地方（如配对出现的语句、嵌套的 IF 语句、类声明定义等）否已正确地缩进？	□
	程序代码布局结构清楚吗？	□
	注释准确并有意义吗？在每一个模块之前，是否有注释说明，描述该模块的输入/输出、参数、功能处理和其调用的外部模块以及该模块是否有使用限制等？	□
	是否有多余的资源定义和宏定义？	□
	头文件是否使用了 ifndef/define/endif 预处理块?	□
	程序结构和模块功能定义清楚吗？	□
	是否遵循该语言的指令编写格式？	□
	注释的行数不少于代码总行数的 1/5 吗？	□
	注释说明和代码功能一致吗？	□
	错误处理分支信息表达清楚吗？	□
	每一个模块单元的圈复杂度都小于 10 吗？	□
	模块内做到了高内聚、模块之间达到了低藕合吗？	□
	模块的扇出不超出 7-9 之间吗？	□
	屏蔽了没有明确含义的输入和按键吗？	□
	常量、变量、类、数据结构等命名有意义吗？	□
函数接口检查	实参和形参的个数、属性和次序一致吗？	□
	对另一个模块的每一次调用：全部所需的参数是否已传送给每一个被调用的模块？被传送的参数值是否正确设置？	□
	函数功能是否齐全？	□
	函数返回值类型正确吗？	□
	return 语句是否返回指向“栈内存”的“指针”或者“引用”？	□
	函数的返回值是否全面反应了各种状态和结果？	□
程序语言检查	动态连接库和外部设备接口驱动程序使用正确吗？	□
	动态分配的指针是否在不使用之后删除，并释放内存？	□
	调用类成员函数或 API 函数时，检查了返回值吗？	□
	文件、数据库和注册表等打开后，在对其进行操作之后是否进行了关闭？	□
	对于使用附带例外的函数是否增加了例外处理程序？如对数据库或文件操作。	□
	变量的数据类型定义是否合理？	□
	程序中是否出现相同的局部变量和全部变量？	□
	数据类型转换使用了正确的转换函数并转换正确吗？	□
	是否使用了只用于调试版本的函数、宏等？	□
	有多个线程的程序中，资源分配是否合理，会不会造成死锁？	□
	在使用 GDI 对象后是否进行删除？	□



检查大项	检查小项	是否
	变量的作用域和生命期是否满足设计的目的？	□
	表达式中运算优先级是否正确？	□
	是否忘记写 switch 的 default 分支？	□
	使用 goto 语句时是否留下隐患？例如跳过了某些对象的构造、变量的初始化、重要的计算等。	□
	Case 语句的结尾是否忘了加 break？	□
	如果有运算符重载，则检查运算符重载是否正确？	□
类检查	类封装是否合理，检查成员函数和成员变量的访问属性是否满足操作要求？	□
	外部可以修改类的行为吗？	□
	内联函数代码足够小吗？	□
	多重继承中，虚拟函数定义明确吗？	□
	继承类和自定义类所封装的函数和过程是否合理？类的功能是否详细，全面？	□
	是否使用了合理的类？查看该类使用时需要注意的问题。	□
	是否违背编程规范而让 C++ 编译器自动为类产生四个缺省的函数：(1) 缺省的无参数构造函数；(2) 缺省的拷贝构造函数；(3) 缺省的析构函数；(4) 缺省的赋值函数。	□
	构造函数中是否遗漏了某些初始化工作？	□
	是否正确地使用构造函数的初始化表？	□
	析构函数中是否遗漏了某些清除工作？	□
	是否错写、错用了拷贝构造函数和赋值函数？	□
	赋值函数一般分四个步骤：(1) 检查自赋值；(2) 释放原有内存资源；(3) 分配新的内存资源，并复制内容；(4) 返回 *this。是否遗漏了重要步骤？	□
	是否违背了继承和组合的规则？ (1) 若在逻辑上 B 是 A 的“一种”，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。 (2) 若在逻辑上 A 是 B 的“一部分”(a part of)，则不允许 B 从 A 派生，而是要用 A 和其它东西组合出 B。	□
内存检查	每一个域在每一次使用前正确地初始化了吗？	□
	是否忘记为数组和动态内存赋初值？（防止将未被初始化的内存作为右值使用）	□
	数组或指针的下标是否越界？	□
	动态内存的申请与释放是否配对？（防止内存泄漏）	□
	是否有效地处理了“内存耗尽”问题？	□
	是否修改“指向常量的指针”的内容？	□
	每个域是否已由正确的变量类型声明？	□
	存储区重复使用吗？可能出现冲突吗？	□
	用 malloc 或 new 申请内存之后，是否立即检查指针值是否为 NULL？（防止使用指针值为 NULL 的内存）	□



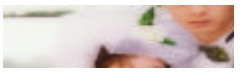
检查大项	检查小项	是否
	是否出现野指针？例如 （1）指针变量没有被初始化。 （2）用 free 或 delete 释放了内存之后，忘记将指针设置为 NULL。	□
	未使用的内存中的内容是否影响系统安全？处理是否得当？	□
测试和转移检查	是否进行了浮点数相等比较？	□
	测试条件逻辑组合正确吗？	□
	逻辑“或”中一个条件满足就执行对其它逻辑表达式有影响吗？	□
	用于测试的是正确的变量吗？	□
	每个转移目标正确并至少执行一次吗？	□
	三种情况（大于 0，小于 0，等于 0）是否已全部测试？边界值是否进行了测试？	□
	循环语句是否有正常跳出循环的条件吗？是否会出现死循环？break 和 continue 语句使用正确吗？	□
性能检查	逻辑是否被最佳地编码？	□
	提供的是一般的错误处理还是异常的例程？	□
	对屏幕输出操作，是否到达了最快的刷新速度？效率是否为最佳？需部分刷新区域的地方是否进行了全部刷新？	□
	有无可优化的程序块、函数或子程序等？	□
	算法是否可以优化？	□
可维护性检查	注释比例达到 25%以上吗？	□
	标号和子程序名符合代码的意义吗？	□
	是否使用了 GOTO 语句？	□
	是否使用了非通用的函数库？对于非标准的库是否提供了源程序？	□
	对于重复出现的常量是否定义了宏？	□
	对于重复出现并完成同样单一功能的一段代码，是否用函数对其进行了封装？	□
	避免过多的使用技巧性编程，如使用，是否作了详细解释说明？	□
	错误或异常信息提示正确吗？	□
逻辑检查	代码是否正确地实现了设计功能？	□
	编码是否做了设计所规定以外的内容？	□
	每个循环是否执行正确的次数？	□
	输入参数的所有异常值是否已直接测试？	□
	逻辑判断表达式符合程序设计吗？	□
软件多余物	有没有不可能执行到的代码？	□
	有没有即使不执行也不影响程序功能的指令？	□
	有没有未引用的变量、标号和常量？	□
	有没有多余的程序单元？	□



附录二：单元测试 Bug 清单

单元测试 Bug 清单

下面由测试人员填写					
项目名称				版本号	
Bug ID		用例 ID		提交时间	
提交人				Email	
提交给				Email	
问题名称					
问题描述					
项目阶段	□需求分析 □详细设计 □集成测试 □验收测试 □结构设计 □单元测试 □系统测试 □维护				
问题类型	□硬件 □设计 □编码 □建议 □疑问				
问题级别	□重大 □高 □中 □低				
可再现否	□是 □否 □不一定				
再现描述					
修改建议					
备 注					
下面由 Bug 管理人员填写					
优先级	□立即解决 □尽快解决 □下一阶段解决 □可能的情况下解决				
意见	对问题解决的意见，建议，修改期限等				
是否纳入 Bug 管理	□ 是 □否				
备 注					
下面由 问题解决者填写					
问题状态	□正在解决 □无法再现问题 □依照设计 □已经解决 □保留 □问题被撤回				
	已解决版本		解决时间		
处理说明					
下面由测试验证人员填写					
验证人			验证版本		
验证时间					
问题状态	□已经解决 □没有解决 □已经解决但引起新的问题				
	已经解决但引起新的问题 Bug ID				
备 注					



附录三：驱动模块（类）模板

一般情况下，应用软件系统每个被测单元由一个 C++类组成，由一些的.h 头文件和.cpp 类实现文件组成。则测试单元通常可以由 3 个文件组成，测试单元头文件，测试单元实现文件和测试主函数文件。假定被测类类名为 DemoClass，测试单元命名为 DemoUnit，如果一个测试单元只测试一个被测类，可以使 DemoUnit 与 DemoClass 一致，则这 3 个文件分别取名为：

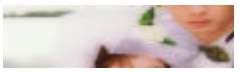
- 测试单元头文件：DemoClassTest.h
- 测试单元实现文件：DemoClassTest.cpp
- 测试主函数文件：DemoUnitMain.cpp

以下以描述这 3 个的框架结构。一个完整的 Demo 可以参考 DemoClass 测试单元[7]。

1) 测试单元头文件

测试单元头文件采用 CppUnit 规范定义测试类，声明测试用例方法。对于被测类 DemoClass，其测试单元头文件取名为 DemoClassTest.h，其结构如下所示：

```
/*  DemoClass 测试代码头文件        */
#include "../DemoClass.h"           /*  包含被测单元的头文件（在上层目录中）     */
#include <cppunit/TestFixture.h>        /*  使用 TestFixture 类  */
#include <cppunit/extensions/HelperMacros.h>    /*  使用 Helper Macros   */
#include <cppunit/TestSuite.h>          /*  使用 TestSuite   类*/

class DemoClassTest : public CppUnit::TestFixture   /*  继承 TestFixture 定义测试类
    */
{
public:
    CPPUNIT_TEST_SUITE( DemoClassTest );    /*  声明 TestSuite 名，与测试类一致  */
    CPPUNIT_TEST( test_tc1 );       /*  在 TestSuite 中添加测试用例  */
    CPPUNIT_TEST( test_tc2 );       /*  在 TestSuite 中添加测试用例  */
    ……                          /*  在 TestSuite 中添加其他测试用例  */
    CPPUNIT_TEST_SUITE_END();       /*  TestSuite 声明结束       */

protected:
    demo_unit *unit1, *unit2, *unit3;       /*  测试过程涉及的被测类对象指针，在
setup()函数中
                                    动态建立并初使化，在 teardown()函数中撤销
    */
    …….
public:
    void setUp();               /*  测试准备或建立测试环境          */
    void tearDown();            /*  测试结束撤销测试环境，如释放动态变量等  */
    void test_tc1();            /*  测试用例方法定义     */
    void test_tc2();            /*  测试用例方法定义     */
```



```
    …….                         /*  其他测试用例方法声明     */
……                              /*  开发者自定义的其他数据成员和方法成员定义     */
}
```

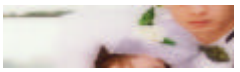
2) 测试单元实现文件

测试单元实现文件实现测试单元头文件中定义的各个测试用例方法和测试类的其他方法成员。对应上述测试单元头文件，相应的测试单元实现文件为 DemoClass.cpp，其结构表示如下：

```
/*  demo unit 测试单元   源代码  */

#include "DemoClassTest.h"  /*  包含 DemoClass 的测试单元头文件  */
#include <string.h>         /*  stl 的 std::string 类     */
#include <iostream.h>       /*  io 流定义头文件              */
#include <cppunit/TestAssert.h> /*  程序中用到了 TestAssert 类   */

/*  在 CppUnit 中注册 DemoClass 的 TestSuite，测试类名一致*/
CPPUNIT_TEST_SUITE_REGISTRATION( DemoClassTest);
void DemoClassTest::setUp()                 /*  建立测试环境        */
{
    unit1 = new DemoClass( 1, 2 );          /*  如创建被测类对象     */
    …..
}

void DemoClassTest::tearDown()              /*  销毁测试环境     */
{
    delete unit1;                           /*  释放被测对象     */
    ……
}

void DemoClassTest::test_tc1()          /*  的测试用例方法 1 的实现  */
{
    /*  测试用例方法的实现代码，测试人员在代码中调用被测模块的方法进行测试，通过 CppUnit 的 ASSERT 宏检查被测模块代码的运行是否正确，并报告异常    */

    /*  执行到测试用例方法的最后，意味在此之前没有发生测试异常事件，意味者本测试用例成功，添加一个语句输出本测试用例信息及其测试成功的信息，其格式为：
    "PASS: <测试用例方法名称>，<测试用例功能描述> <换行符>  */
    cout<<"PASS: test_tc1，测试 DEMO CLASS 的构造函数正确性\n";
}
……                  /*  最后添加测试类其他方法的实现     */
```

3) 测试主函数文件

测试单元主函数用于执行测试类中定义的各种测试用例方法，执行各个测试用例，如果我们充分利用 CppUnit 提供的宏来书写测试单元，测试单元主函数可以设计成与被测模块和测试类无关，而对所有被测模块使用同一个测试主函数文件。

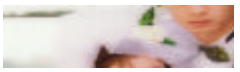


被测模块 DemoClass 的测试驱动程序文件名为：DemoUnitMain.cpp。该程序的内容如下：

```
/*  单元测试主函数，固定不变    */

#include <cppunit/TextTestResult.h>
#include <cppunit/TestSuite.h>
#include <vector>
#include <iostream>
#include <cppunit/ui/text/TestRunner.h>
#include <cppunit/TestCaller.h>
#include <cppunit/TestCase.h>

#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/CompilerOutputter.h>

#include <vector>
#include <iostream>

int main( int argc, char **argv)
{
    CppUnit::TextUi::TestRunner runner;
    CppUnit::TestFactoryRegistry                    &registry                    =
CppUnit::TestFactoryRegistry::getRegistry();
    runner.addTest( registry.makeTest() );      /*  加载测试类实现文件中注册的
TestSuite   */
    bool wasSucessful = runner.run( "", false );    /*  运行所有 TestSuite      */
    return wasSucessful ? 0 : 1;
}
```

4) 编程规范

以上从 CppUnit 测试工具的特点列出测试单元的程序结构，每个 C/C++测试程序文件应的编程风格和规范应该遵从应用软件编程规范[6]的要求。



附录四：单元测试实例介绍

DemoUnit 示例测试单元介绍

该 DemoUnit 包括 DemoClass 和 Complex 两个被测类。DemoUnit 中按照上述规范设计了 DemoClass 的测试类代码，但 Complex 的测试类并没有按以上规范设计。

在 PC Linux 平台下执行测试

在 TestDemo/PC Linux/Makefile 文件实现了在 PC Linux 下执行测试和生成各类测试报告的脚本。测试执行方法介绍如下：

1) 执行正常测试，查找模块缺陷，显示测试结果

   make default -i

2) 生成正确性测试报告.log 文件和代码覆盖率文件

   make gcov -i OPT=-O0

   make gcov -i OPT=-O2

   make gcov -i OPT=-O3

3) 内存溢出分析

   make ccmalloc -i

4) 代码性能分析

   make -i gprof

5) 删除各种中间临时文件

   make clean