

# 如何使用 C++test 工具满足逻辑覆盖

## 目 录

逻辑覆盖.....	2
Parasoft C++test 工具.....	2
C++test 覆盖率类型.....	3
行覆盖.....	3
语句覆盖.....	4
块覆盖.....	5
路径覆盖.....	6
判断（分支）覆盖.....	7
修正的条件/判断覆盖(MC/DC).....	7
简单条件覆盖.....	9
函数覆盖.....	10
总结.....	10

## 逻辑覆盖

逻辑覆盖是白盒测试的一种覆盖标准。白盒测试法的覆盖标准有逻辑覆盖、循环覆盖和基本路径测试。其中逻辑覆盖包括语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖和路径覆盖。

## Parasoft C++test 工具

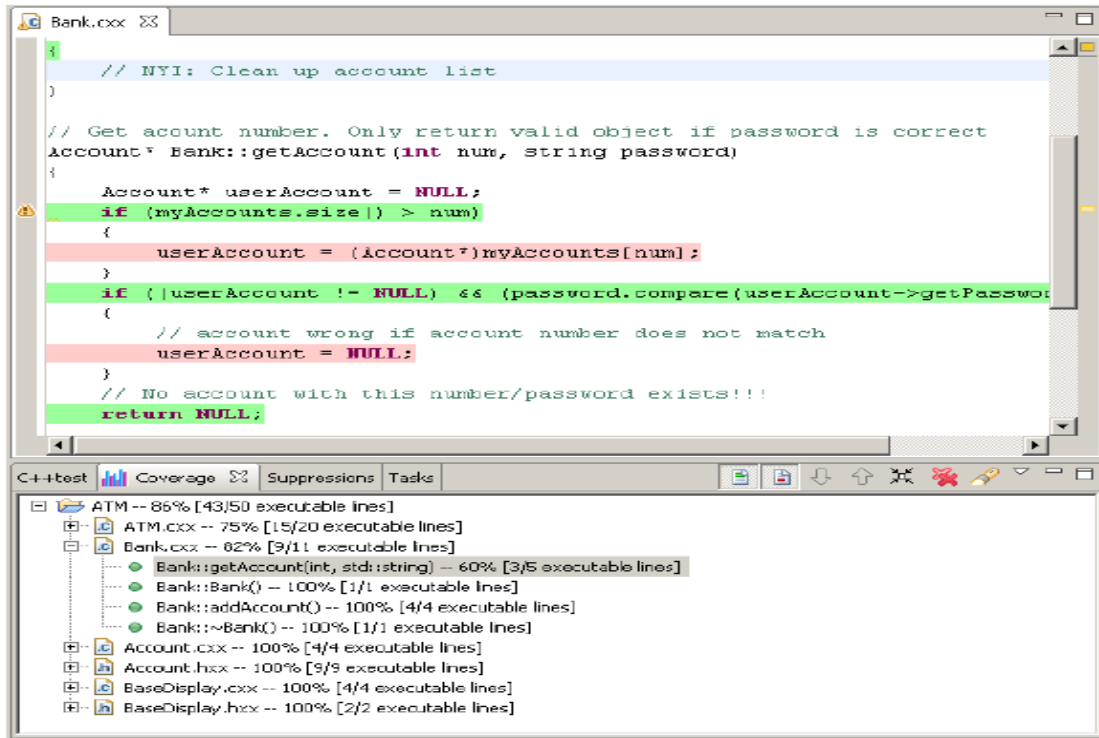
Parasoft C++test 是美国公司 Parasoft 开发的一款专业白盒测试工具。C++test 能够使团队开发更好的代码，实施更有效的测试，以及持续地监视实现其质量目标的过程。使用 C++test 经时间验证的重要最佳实践--例如静态分析，全面的代码审查，运行时错误检测，集成覆盖率分析的单元测试和组件测试能够在开发周期的开始阶段，自动地在开发者的桌面上完成。能够通过命令行模式自动化地执行回归和持续集成测试，为监测和分析质量趋势提供数据。针对嵌入式和跨平台开发，C++test 可以用在基于宿主环境和目标环境的代码分析和测试流程中。

下面的章节内容重点描述了 C++test 所包含的覆盖率类型以及如何使用 C++test 工具满足逻辑覆盖要求。

## C++test 覆盖率类型

### 行覆盖

定义：指出多少源码的可执行行受控制流影响至少一次。如果所有可执行行受到影响至少一次，将会获得完全的 100% 行覆盖率。



## 语句覆盖

定义：表示有多少可执行源代码语句是控制流至少一次能过到达的。如果所有的可执行语句至少到达一次，将获得 100% 语句覆盖率。

```
35 void ATM::fillUserRequest(UserRequest request, double amount)
36 {
37
38     if (myCurrentAccount)
39         switch (request)
40         {
41             case REQUEST_BALANCE:
42                 showBalance(); break;
43             case REQUEST_DEPOSIT:
44                 makeDeposit(amount); break;
45             case REQUEST_WITHDRAW:
46                 withdraw(amount); break;
47         }
48 }
```

00 %

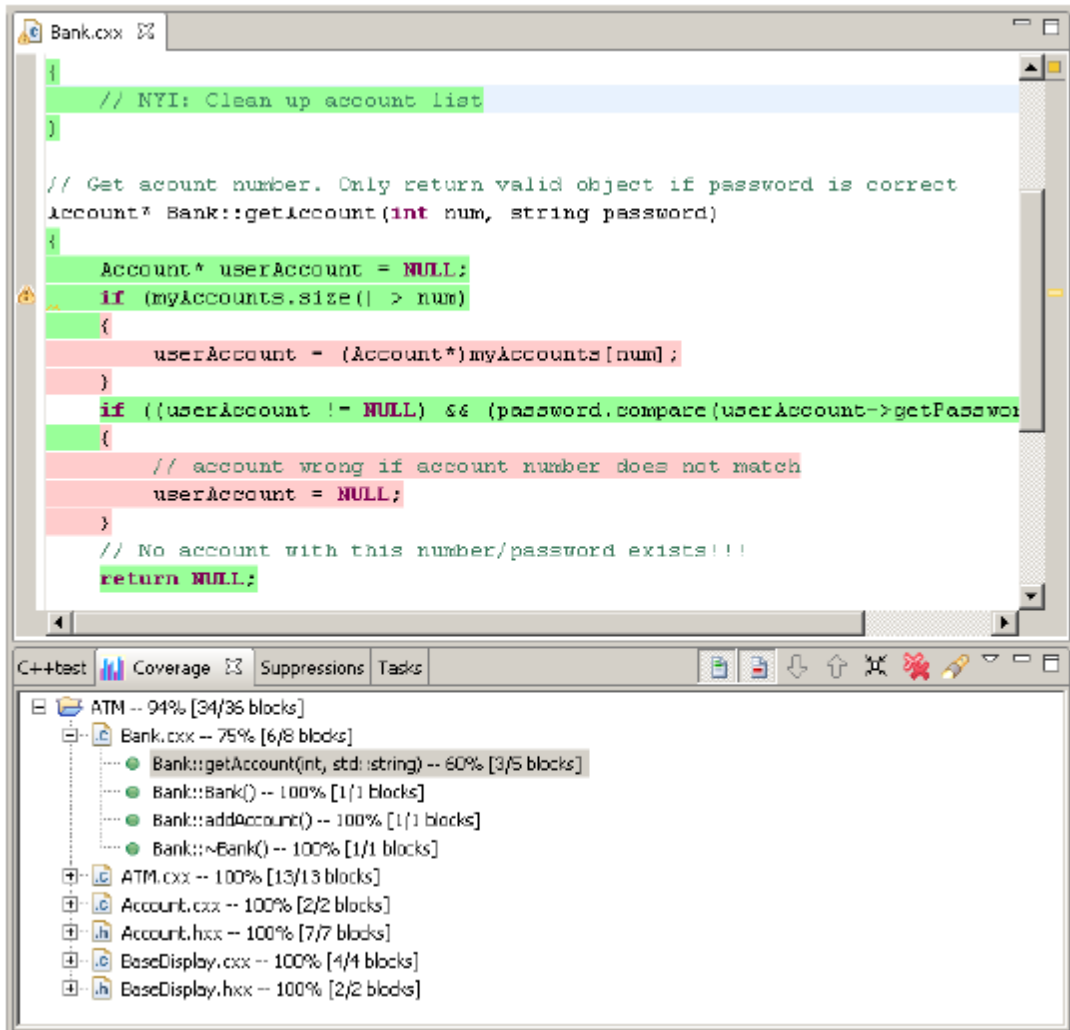
coverage

Statement Coverage 25% [29/118 statements]

- ATM - 25% [29/118 statements]
  - Source Files - 24% [26/107 statements]
    - ATM.cpp - 0% [0/62 statements]
    - Account.cxx - 25% [1/4 statements]
    - Bank.cxx - 50% [6/12 statements]
    - ATM.cxx - 65% [17/26 statements]
      - ATM::makeDeposit(double) - 33% [1/3 statements]
      - ATM::showBalance() - 33% [1/3 statements]
      - ATM::withdraw(double) - 33% [1/3 statements]
      - ATM::fillUserRequest(ATM::UserRequest, double) - 63% [5/8 statements]

## 块覆盖

定义：与行覆盖率类似（除了带有块覆盖率的测量代码单元是一个基本块），表示源码中有多少基本块受到控制流影响至少一次。



```
Bank.cpp
1
// NYI: Clean up account list
}

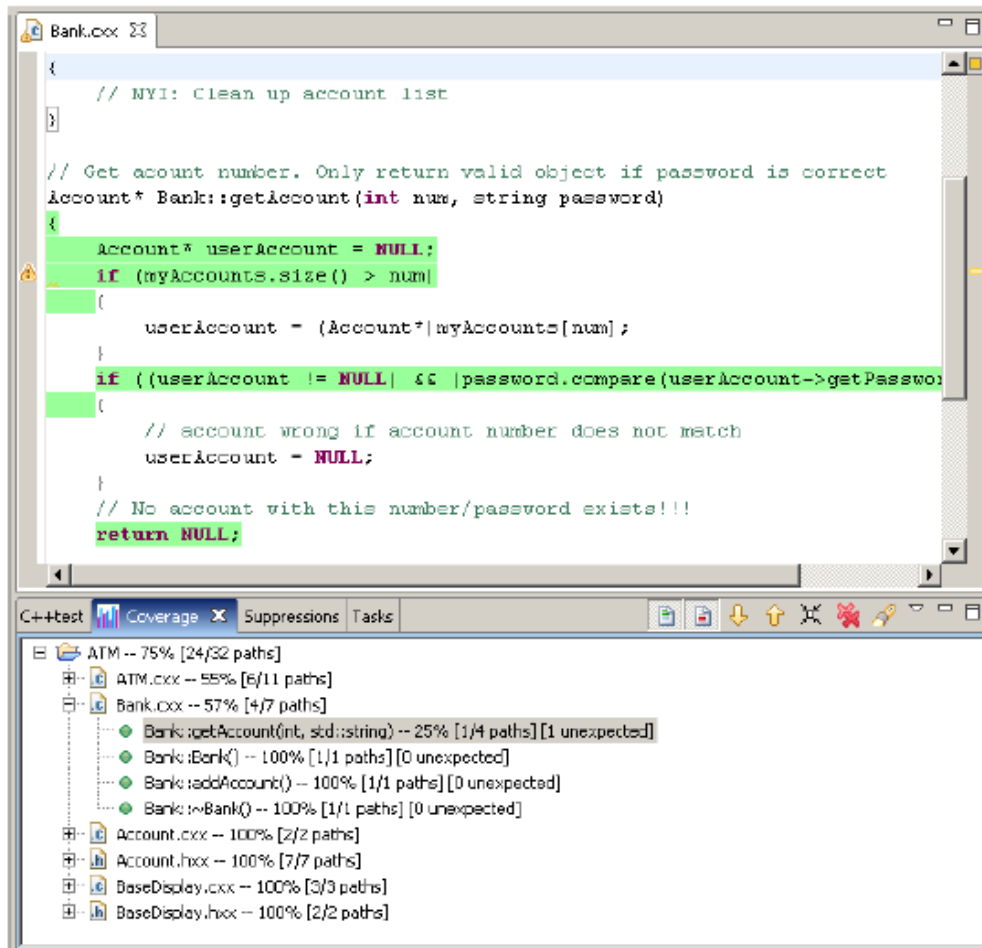
// Get account number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPassword() == 0))
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
}
```

C++test Coverage Suppressions Tasks

- ATM -- 94% [34/36 blocks]
- Bank.cpp -- 75% [6/8 blocks]
  - Bank::getAccount(int, std::string) -- 60% [3/5 blocks]
  - Bank::Bank() -- 100% [1/1 blocks]
  - Bank::addAccount() -- 100% [1/1 blocks]
  - Bank::~Bank() -- 100% [1/1 blocks]
- ATM.cpp -- 100% [13/13 blocks]
- Account.cpp -- 100% [2/2 blocks]
- Account.hxx -- 100% [7/7 blocks]
- BaseDisplay.cpp -- 100% [4/4 blocks]
- BaseDisplay.hxx -- 100% [2/2 blocks]

## 路径覆盖

定义：表明是否一个给定的函数中每条可能的路径随后有控制流。



```
Bank.cpp
{
    // NYI: Clean up account list
}

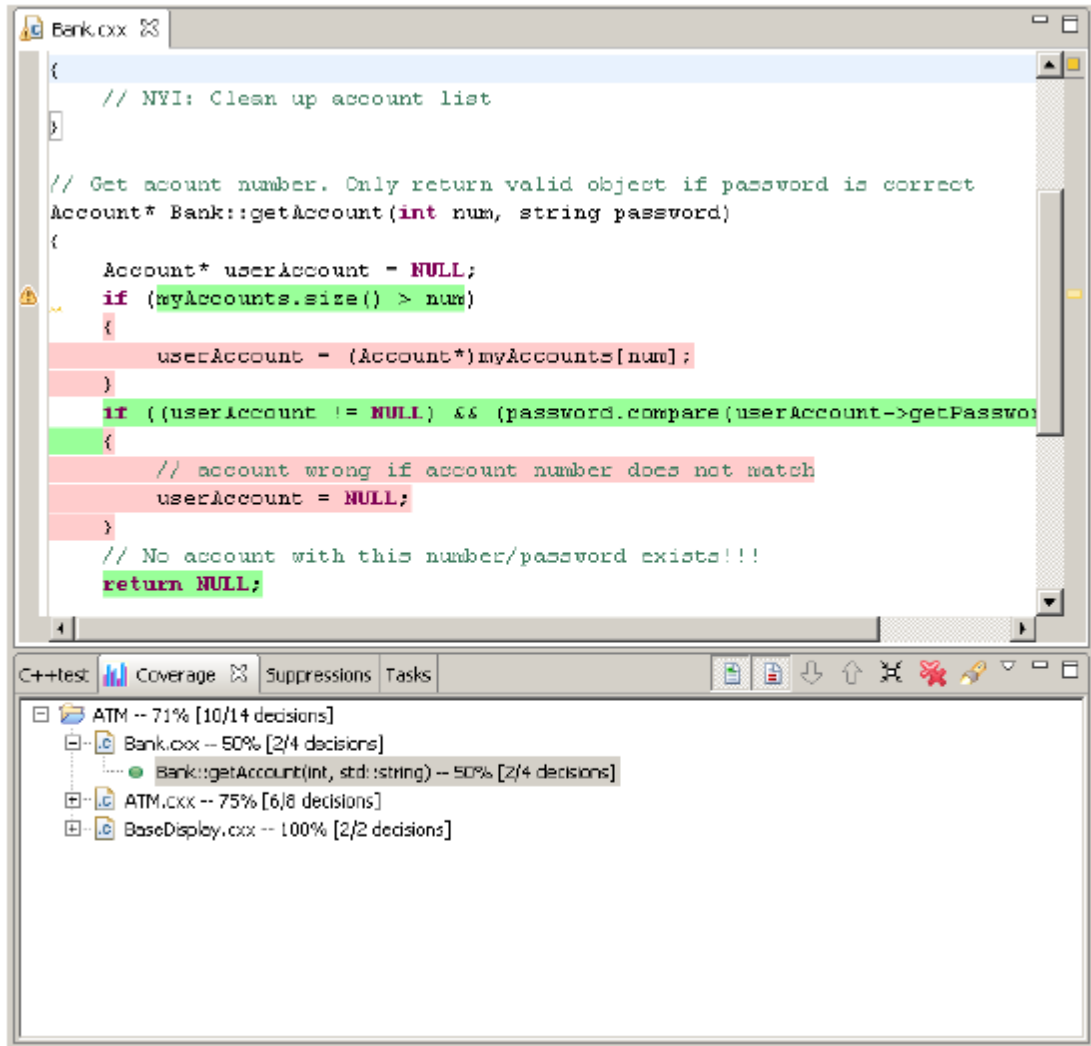
// Get account number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if (userAccount != NULL && !password.compare(userAccount->getPasswo
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
}
```

C++test Coverage X Suppressions Tasks

- ATM -- 75% [24/32 paths]
- ATM.cpp -- 55% [6/11 paths]
- Bank.cpp -- 57% [4/7 paths]
  - Bank::getAccount(int, std::string) -- 25% [1/4 paths] [1 unexpected]
  - Bank::Bank() -- 100% [1/1 paths] [0 unexpected]
  - Bank::addAccount() -- 100% [1/1 paths] [0 unexpected]
  - Bank::~Bank() -- 100% [1/1 paths] [0 unexpected]
- Account.cpp -- 100% [2/2 paths]
- Account.hxx -- 100% [7/7 paths]
- BaseDisplay.cpp -- 100% [3/3 paths]
- BaseDisplay.hxx -- 100% [2/2 paths]

## 判断（分支）覆盖

定义：表明源码中有多少分支控制流通过。当每一个判决在所有的分支点取得所有可能的结果至少一次时，可获取完整的，100% 覆盖。



```
Bank.cpp
{
    // NYI: Clean up account list
}

// Get account number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPasswo
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
}
```

C++test Coverage Suppressions Tasks

- ATM -- 71% [10/14 decisions]
- Bank.cpp -- 50% [2/4 decisions]
- Bank::getAccount(int, std::string) -- 50% [2/4 decisions]
- ATM.cpp -- 75% [6/8 decisions]
- BaseDisplay.cpp -- 100% [2/2 decisions]

## 修正的条件/判断覆盖(MC/DC)

定义：MC/DC 与国际技术标准 DO-178B (RTCA) 一致，此标准详细说明了软件证明的标准，包括实时嵌入式系统，危急任务设施和航空工业的系统。根据 DO-178B 标准必需满足下面三个条件才能获得全部的 (100%) MC/DC 覆盖率：

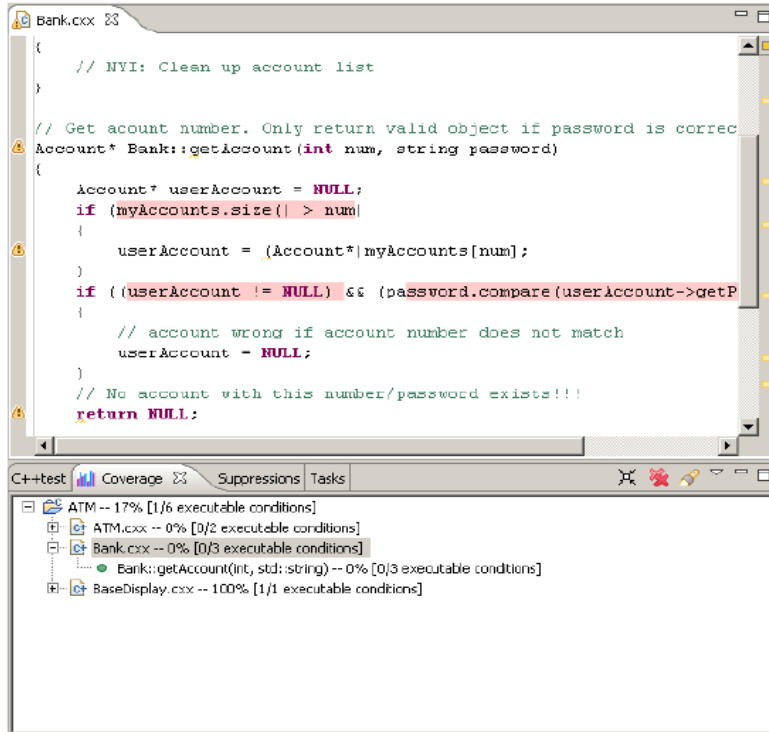
- a) 每个判断至少有一次已经产生所有可能的结果
- b) 判断中的每一个条件至少有一次已经产生所有可能的结果。
- c) 判断中每一个条件已经证明独立地影响判断结果。

由于 C++test 认为每一个条件和判断可能只有两个 MC/DC 覆盖率的结果--真或假 -- C++test 只检查上面刚刚列出的第三个选项 (c)，因为 (c) 中暗含条件 (a) 和 (b)。

通过改变一些特殊条件，又固定所有其他可能的条件，来证明这种条件能够独立地影响判决结果，因此，为了测试某个给定的条件，C++test 在下面这些地方寻找测试用例：

- 测试条件有真和假两种结果
- 判断中的其他情况不改变（或者 C/C++ 中的算子逻辑短路）
- 判断结果发生改变

因此，为了计算 MD/DC 率，C++test 使用下面的公式： $MC/DC = m/n$ ，这里  $m$  指独立影响判决结果的布尔条件数， $n$  指判决中条件的总数。



```
Bank.cxx
{
    // NYI: Clean up account list
}

// Get account number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getP
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
}
```

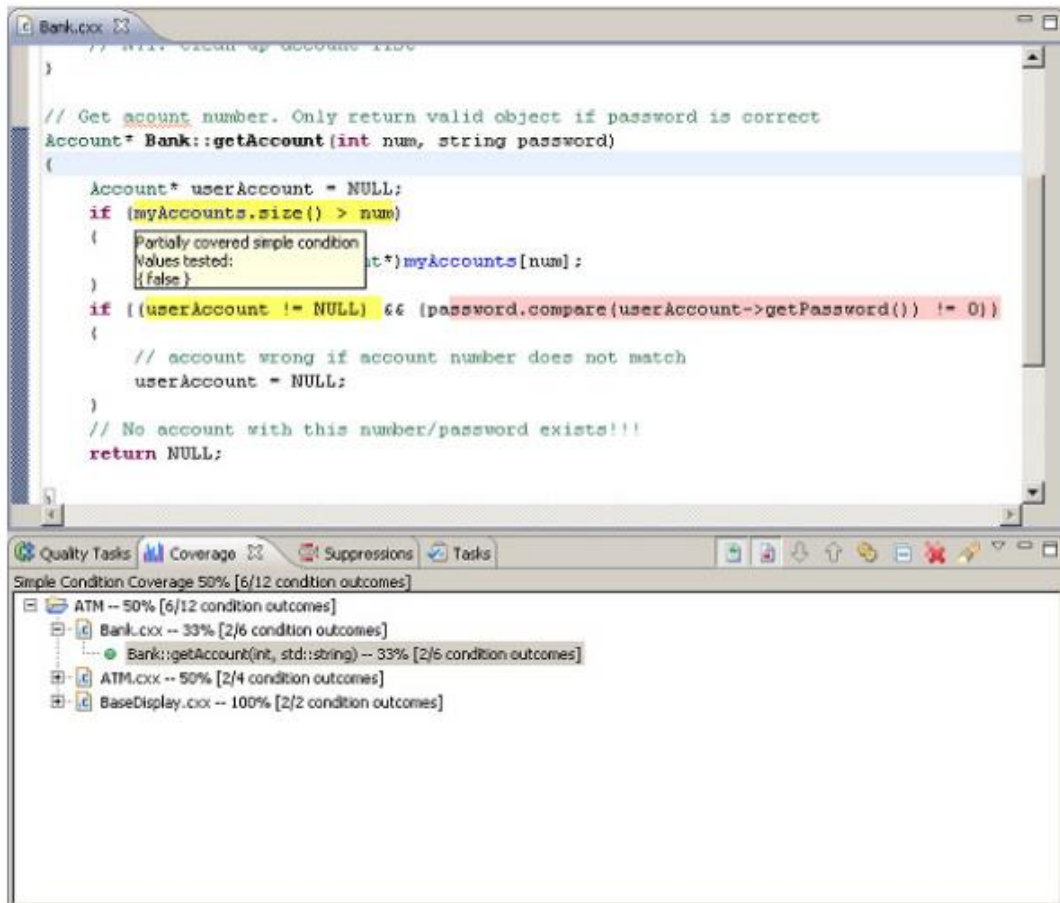
C++test Coverage Suppressions Tasks

- ATM -- 17% [1/6 executable conditions]
- ATM.cxx -- 0% [0/2 executable conditions]
- Bank.cxx -- 0% [0/3 executable conditions]
- Bank::getAccount(int, std::string) -- 0% [0/3 executable conditions]
- BaseDisplay.cxx -- 100% [1/1 executable conditions]



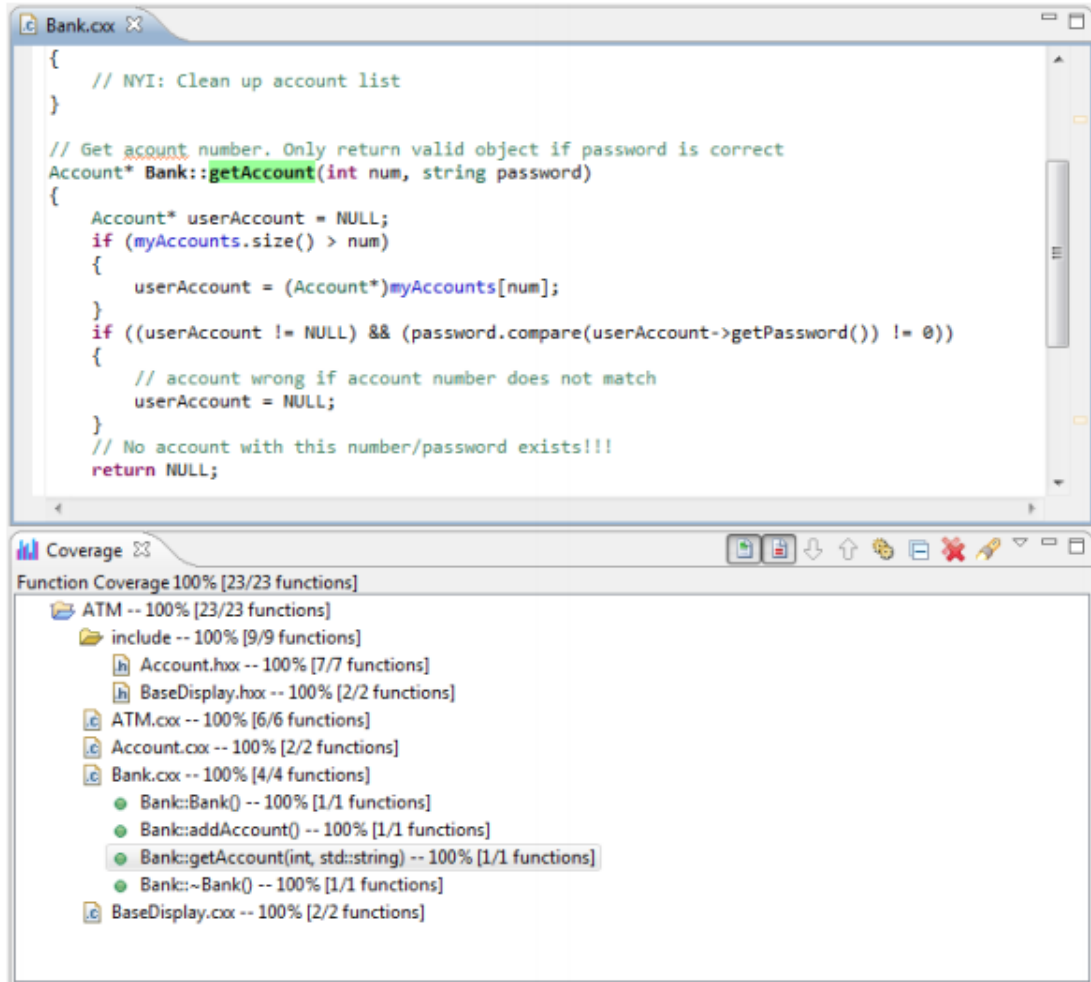
## 简单条件覆盖

定义：指明所有判断条件的结果覆盖率。判断结果的总数等于  $2 * n$ ，这里  $n$  指判断条件的数目。因此，为了获取 100% 的覆盖面，所有的条件必需获得所有可能的结果。



## 函数覆盖

定义：一次执行后源码中有多少函数被至少执行了一次。如果所有的函数都至少执行了一次，那么覆盖率就可以达到 100% 。



```
Bank.cpp
{
    // NYI: Clean up account list
}

// Get account number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
    Account* userAccount = NULL;
    if (myAccounts.size() > num)
    {
        userAccount = (Account*)myAccounts[num];
    }
    if ((userAccount != NULL) && (password.compare(userAccount->getPassword()) != 0))
    {
        // account wrong if account number does not match
        userAccount = NULL;
    }
    // No account with this number/password exists!!!
    return NULL;
}
```

Coverage 100% [23/23 functions]

- ATM -- 100% [23/23 functions]
  - include -- 100% [9/9 functions]
    - Account.hxx -- 100% [7/7 functions]
    - BaseDisplay.hxx -- 100% [2/2 functions]
  - ATM.cpp -- 100% [6/6 functions]
  - Account.cpp -- 100% [2/2 functions]
  - Bank.cpp -- 100% [4/4 functions]
    - Bank::Bank() -- 100% [1/1 functions]
    - Bank::addAccount() -- 100% [1/1 functions]
    - Bank::getAccount(int, std::string) -- 100% [1/1 functions]
    - Bank::~Bank() -- 100% [1/1 functions]
  - BaseDisplay.cpp -- 100% [2/2 functions]

## 总结

通过以上分析和说明，可以看到作为白盒测试的专业工具 Parasoft C++test 完全支持和满足逻辑覆盖的要求。