选择题

2154312 郑博远

1. （ B ）在软件开发的过程中，若能推迟暴露其中的错误，则为修复和改正错误所花费的代价就会降低。

   A. 真

   B. 假

2. （ A ）好的测试是用少量测试用例运行程序，发现被测程序尽可能多的错误。

   A. 真

   B. 假

3. （ B ）好的测试用例应能证明软件是正确的。

   A. 真

   B. 假

4. （ B ）白盒测试仅与程序的内部结构有关，完全可以不考虑程序的功能要求。

   A. 真

   B. 假

5. （ A ）等价类划分方法将所有可能的输入数据划分成若干部分，然后从每一部分中选取少数有代表性的数据作为测试用例。

   A. 真

   B. 假

6. 使用独立测试团队的最好理由是（ C ）。

   A. 软件开发人员不需要做任何测试

   B. 测试人员在测试开始之前不参与项目

   C. 测试团队将更彻底地测试软件

   D. 开发人员与测试人员之间的争论会减少

7. 类的行为应该基于（ D ）进行测试。

A. 数据流图

B. 用例图

C. 对象图

D. 状态图

8. 下面的（ ABDE ）说法是正确的。

A. 恢复测试是以各种方式迫使软件失效从而检测软件是否能够继续执行的一种系统测试。

B. 安全测试是检测系统中的保护机制是否可以保护系统免受非正常的攻击。

C. 压力测试是检测在极限环境中使用系统时施加在用户上的压力。

D. 功能测试是根据软件需求规格说明和测试需求列表，验证产品的功能实现是否符合需求规格。

E. 安装测试是保证应用程序能够被成功地安装。

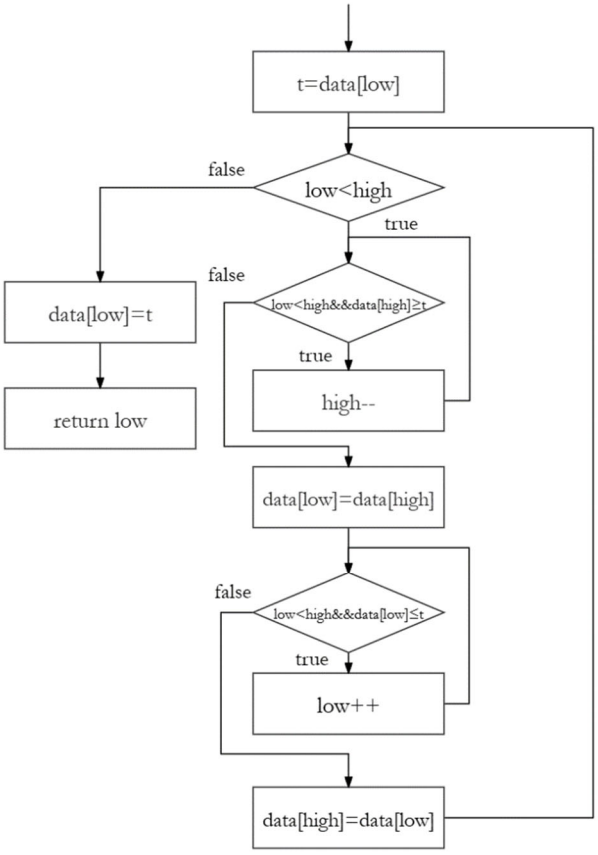
作业

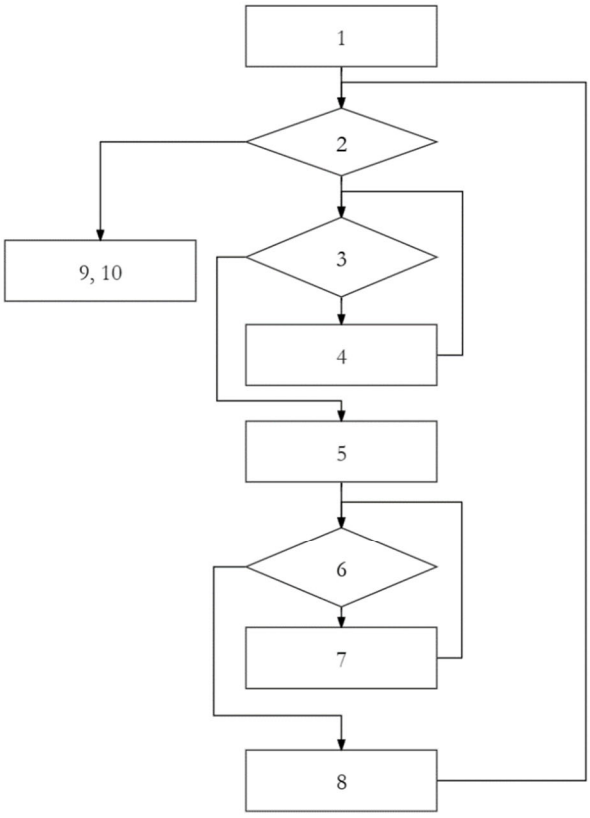
1. 理解下面的程序结构，请使用基本路径方法设计该程序测试用例。

```
#include <stdio.h>
int partition(int *data, int low, int high)
{
 0  int t = 0;

 1  t = data[low];
 2  while (low < high) {
 3    while (low < high && data[high] >= t)
 4      high--;
 5    data[low] = data[high];
 6    while(low < high && data[low] <= t)
 7      low++;
 8    data[high] = data[low];
    }
 9  data[low] = t;

10 return low;
}
```





环路复杂度 V(G) = 11-9+2 = 3+1 = 4

基本路径集：

P1：1→2→3→4→5→6→7→8→9

P2：1→2→3→4→5→6→8→9

P3：1→2→3→5→6→7→8→9

P4：1→9

路径	输入	预期结果
P1	data=[2,1,0,4],low=0,high=1	data=[0,1,2,4],return 2
P2	data=[0,1],low=0,high=1	data=[0,1], return 0
P3	data=[1,0],low=0,high=1	data=[0,1], return 1
P4	data=[1],low=0,high=0	data=[1], return 0

2. 请使用等价类划分和边界值分析的方法， 给出 getNumDaysInMonth(int month, int year)方法的测试用例，其中 getNumDaysInMonth 方法根据给定的月份和年份返回该月份的总天数。

（1）请给出划分的等价类。

	有效等价类	无效等价类
月份	2月（28/29天）	非正数
	30天的月份	大于12
	31天的月份	
年份	平年	非正数
	闰年	

(2) 测试用例设计：

序号	输入参数	期望输出	备注
1	(2,1900)	28	2月，（世纪）平年
2	(2,2000)	29	2月，（世纪）闰年
3	(4,2023)	30	有30日的月，平年
4	(4,2024)	30	有30日的月，闰年
5	(12,2007)	31	有31日的月，平年
6	(12,1996)	31	有31日的月，闰年
7	(-1,2002)	无效	非正数月，平年
8	(0,1908)	无效	非正数月，闰年
9	(-2,-4)	无效	非正数月，非正数年
10	(100,3000)	无效	大于12月，（世纪）平年
11	(13,4000)	无效	大于12月，（世纪）闰年
12	(25,-1)	无效	大于12月，非正数年
13	(2,-2048)	无效	2月，非正数年
14	(4,-19)	无效	有30日的月，非正数年
15	(5,0)	无效	有31日的月，非正数年

3. 现在要对一个咨询公司的薪酬支付软件进行功能测试，该软件的规格说明如下：

对于每周工作超过 40 小时的咨询人员，前 40 小时按照他们的小时薪酬计算，多余的小时数按照双倍小时薪酬计算；对于每周工作不足 40 小时的咨询人员，按照他们的小时薪酬计算，并生成一份缺勤报告；对于每周工作超过 40 小时的长期工作人员，直接按照他们的固定工资计算。

（1） 使用决策表（格式如下）描述上述的薪酬支付规则；

规则		1	2	3
条件桩	每周工作是否超过40小时	是	否	是
	工作人员身份	咨询人员	咨询人员	长期工作人员
动作桩	薪酬计算	前40小时时薪多余双倍	小时时薪	固定工资
	缺勤报告	不生成	生成	不生成

（2） 根据上述决策表，使用下表列出自己设计的测试用例。

序号	输入参数	期望输出
1	咨询人员，每周工作60小时，时薪100元	薪酬8000元
2	咨询人员，每周工作20小时，时薪100元	薪酬2000元，生成缺勤报告
3	长期工作人员，固定工资5000元	薪酬5000元

4. 下面给出的是购买车票（PurchaseTicket） 用例的正常流， 除此之外还应包括无零钱找（NoChange）、缺票（OutofOrder）、超时（TimeOut）和取消（Cancel）等四个备选流，请结合场景法和其他方法设计测试用例。

用例名称	购买车票
前置条件	乘客站在售票机前，有足够的钱买车票。

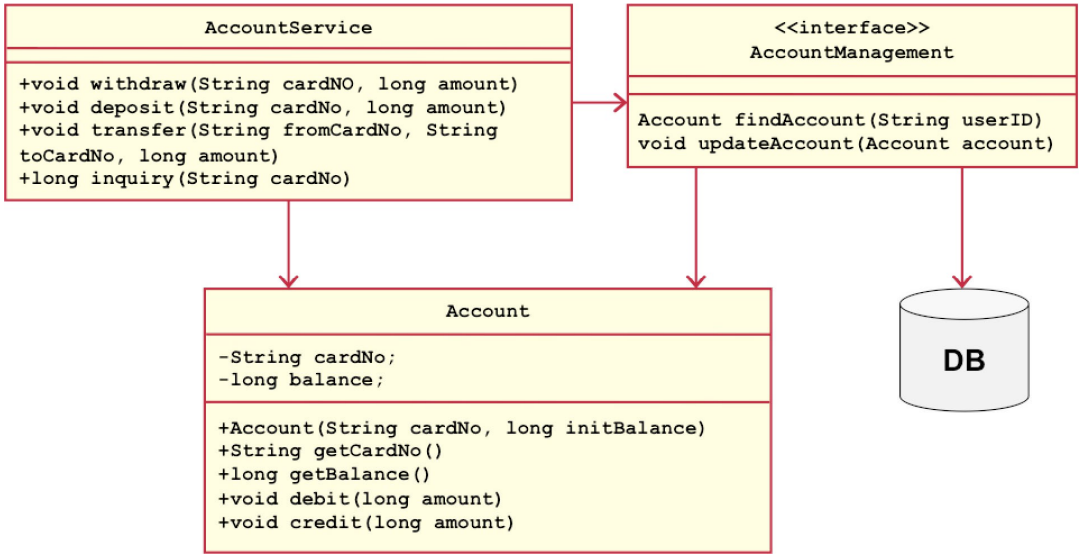
正常流	1. 乘客选择需要达到的地点，如果按下了多个地点按钮，售票机只考虑最后一次按下的地点。 2. 售票机显示出应付款数。 3. 乘客投入钱。 4. 如果乘客在投入足够的钱之前选择了新的地点，售票机应该把所有的钱退还乘客。 5. 如果乘客投入的钱比应付款多，售票机应该退出多余的零钱。 6. 售票机给出车票。 7. 乘客拿走找零和车票。
后置条件	乘客买到了他选择的车票。

要求：使用下表列出自己设计的测试用例。

测试用例编号	测试标题	预置条件	操作步骤	预期输出
1	正常购票流程	乘客站在售票机前，有足够钱买车票	1．选择地点 2．投入足够的钱 3．拿走找零和车票	售票机显示应付款数，成功出票，乘客拿到车票和找零
2	无零钱找	乘客站在售票机前，钱不够找零	1．选择地点 2．乘客投入钱	售票机显示错误信息，提示无零钱找并退款
3	缺票	售票机内无票	1．选择地点 2．投入足够的钱	售票机显示无票信息，不允许乘客选择地点
4	超时	乘客操作超时	1．选择地点 2．长时间不投入钱	售票机取消交易，提示超时
5	取消	乘客决定不购票	1．选择地点 2．选择取消交易	售票机取消交易，退还所有投入的钱
6	多次选择地点后投入钱	乘客多次选择地点	1．多次选择地点 2．最后一次选择后投入足够的钱	售票机按最后一次选择的地点出票，成功出票
7	投入钱后选择新地点	乘客先投入钱，后选择新地点	1．投入钱 2．选择新地点	售票机退还所有投入的钱，按照新的地点计算
8	投入钱大于应付款	乘客投入的钱超过票价	1．选择地点 2．投入超过应付款的钱	售票机成功出票并找零

作业

下面的 UML 图显示了银行卡账户的服务功能：



要求：

（1） 请使用Java 或 C++语言和测试驱动开发方法编写上述程序，并进行单元测试和代码覆盖分析；

（2） 如果银行卡的转账服务增加一个新的需求“一次转账金额限定为 3000 元”，请修改代码并重新进行单元测试和覆盖分析。

（1）

1. Account.h

```
#ifndef ACCOUNT_H
#define ACCOUNT_H

#include <string>

class Account {
private:
    std::string cardNo;
    long balance;

public:
    Account(std::string cardNo, long initBalance);
    std::string getCardNo();
    long getBalance();
    void debit(long amount);
    void credit(long amount);
};

#endif // ACCOUNT_H
```

2. Account.cpp

```
#include "Account.h"

Account::Account(std::string cardNo, long initBalance) : cardNo(cardNo),
balance(initBalance) {}

std::string Account::getCardNo() {
    return cardNo;
}

long Account::getBalance() {
    return balance;
}

void Account::debit(long amount) {
    if (amount <= balance) {
        balance -= amount;
    }
    else {
        throw std::runtime_error("Insufficient funds");
    }
}

void Account::credit(long amount) {
    balance += amount;
}
```

3. AccountService.h

```
#ifndef ACCOUNTSERVICE_H
#define ACCOUNTSERVICE_H

#include "AccountManagement.h"

class AccountService {
private:
    AccountManagement& accountManagement;

public:
    AccountService(AccountManagement& accountManagement);
    void withdraw(std::string cardNo, long amount);
    void deposit(std::string cardNo, long amount);
    void transfer(std::string fromCardNo, std::string toCardNo, long
amount);
    long inquiry(std::string cardNo);
};

#endif // ACCOUNTSERVICE_H
```

4. AccountService.cpp

```
#include "AccountService.h"

AccountService::AccountService(AccountManagement& accountManagement) :
accountManagement(accountManagement) {}

void AccountService::withdraw(std::string cardNo, long amount) {
    Account* account = accountManagement.findAccount(cardNo);
    account->debit(amount);
    accountManagement.updateAccount(account);
}

void AccountService::deposit(std::string cardNo, long amount) {
    Account* account = accountManagement.findAccount(cardNo);
    account->credit(amount);
    accountManagement.updateAccount(account);
}

void AccountService::transfer(std::string fromCardNo, std::string
toCardNo, long amount) {
    Account* fromAccount = accountManagement.findAccount(fromCardNo);
    Account* toAccount = accountManagement.findAccount(toCardNo);
    fromAccount->debit(amount);
    toAccount->credit(amount);
    accountManagement.updateAccount(fromAccount);
    accountManagement.updateAccount(toAccount);
}

long AccountService::inquiry(std::string cardNo) {
    Account* account = accountManagement.findAccount(cardNo);
    return account->getBalance();
}
```

(5) AccountManagement.h

```
#ifndef ACCOUNTMANAGEMENT_H
#define ACCOUNTMANAGEMENT_H

#include "Account.h"
#include <string>

class AccountManagement {
public:
    virtual Account* findAccount(std::string cardNo) = 0;
    virtual void updateAccount(Account* account) = 0;
    virtual ~AccountManagement() = default;
};

#endif // ACCOUNTMANAGEMENT_H
```

单元测试：

```
#include "gtest/gtest.h"
#include "AccountService.h"

class MockAccountManagement : public AccountManagement {
public:
    std::unordered_map<std::string, Account> accounts;

    Account* findAccount(const std::string& cardNo) override {
        if (accounts.find(cardNo) != accounts.end()) {
            return &accounts[cardNo];
        }
        return nullptr;
    }

    void updateAccount(Account* account) override {
        accounts[account->getCardNo()] = *account;
    }

    void addAccount(const std::string& cardNo, long initBalance) {
        accounts[cardNo] = Account(cardNo, initBalance);
    }
};

TEST(AccountTest, Initialization) {
    Account account("12345", 1000);
    EXPECT_EQ(account.getCardNo(), "12345");
    EXPECT_EQ(account.getBalance(), 1000);
}

TEST(AccountTest, DebitCredit) {
    Account account("12345", 1000);
    account.debit(200);
    EXPECT_EQ(account.getBalance(), 800);
    account.credit(300);
    EXPECT_EQ(account.getBalance(), 1100);
}

TEST(AccountServiceTest, Deposit) {
    MockAccountManagement mockStorage;
    AccountService service(mockStorage);
    mockStorage.addAccount("12345", 1000);
    service.deposit("12345", 500);
    EXPECT_EQ(service.inquiry("12345"), 1500);
}

TEST(AccountServiceTest, Withdraw) {
    MockAccountManagement mockStorage;
    AccountService service(mockStorage);
    mockStorage.addAccount("12345", 1000);
```

```
    service.withdraw("12345", 300);
    EXPECT_EQ(service.inquiry("12345"), 700);
}

TEST(AccountServiceTest, Transfer) {
    MockAccountManagement mockStorage;
    AccountService service(mockStorage);
    mockStorage.addAccount("12345", 1000);
    mockStorage.addAccount("67890", 2000);
    service.transfer("12345", "67890", 200);
    EXPECT_EQ(service.inquiry("12345"), 800);
    EXPECT_EQ(service.inquiry("67890"), 2200);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

检查代码覆盖：

Account构造函数、getCardNo、getBalance、debit、credit方法均被覆盖；

AccountService构造函数、withdraw、deposit、transfer、inquiry方法均被覆盖；

每个方法中没有分支语句，因此每个语句都被覆盖。

（2）

修改**AccountService.h**：

```
void AccountService::transfer(std::string fromCardNo, std::string
toCardNo, long amount) {
    if (amount > 3000) {
        throw std::invalid_argument("Transfer amount exceeds the limit of
3000");
    }
    Account* fromAccount = accountManagement.findAccount(fromCardNo);
    Account* toAccount = accountManagement.findAccount(toCardNo);
    fromAccount->debit(amount);
    toAccount->credit(amount);
    accountManagement.updateAccount(fromAccount);
    accountManagement.updateAccount(toAccount);
}
```

在原有基础上添加单元测试：

```
TEST(AccountServiceTest, TransferWithinLimit) {
    MockAccountManagement mockStorage;
    AccountService service(mockStorage);
    mockStorage.addAccount("12345", 1000);
    mockStorage.addAccount("67890", 2000);
    service.transfer("12345", "67890", 2000); // 测试在 3000 元限制内的转账
    EXPECT_EQ(service.inquiry("12345"), 0);
    EXPECT_EQ(service.inquiry("67890"), 4000);
}

TEST(AccountServiceTest, TransferExceedsLimit) {
    MockAccountManagement mockStorage;
    AccountService service(mockStorage);
    mockStorage.addAccount("12345", 1000);
    mockStorage.addAccount("67890", 2000);
    EXPECT_THROW(service.transfer("12345", "67890", 3500),
std::invalid_argument); // 测试超过 3000 元限制的转账
}
```

代码覆盖：

Account构造函数、getCardNo、getBalance、debit、credit方法均被覆盖；

AccountService构造函数、withdraw、deposit、transfer、inquiry方法均被覆盖（transfer中3000元限制对应的两种情况均被覆盖）。