Chapter 3.6 Problem Set

# Problem 3.6.1: Grocery List



For the above UML class diagram, please do the following:

1. Identify three methods which could benefit from an integration test. List them in order from the most beneficial to the least.

The three methods that would benefit most from integration testing are totalQuantity(), addItem(), and find(). The totalQuantity() method is the most beneficial to test because it depends on the interaction between the GroceryList class and multiple Item subclasses, ensuring that all quantities are correctly summed. The addItem() method is also important because it integrates adding different Item objects to the list, testing whether the list correctly stores and manages them. Finally, the find() method benefits from integration testing because it relies on calling getName() from various Item subclasses to determine if an item exists in the list.

1. Create four test cases for one of the methods.

void test\_totalQuantity() {

// Test Case 1: Empty list

GroceryList list1;

assert(list1.totalQuantity() == 0);

// Test Case 2: One regular item

GroceryList list2;

Item\* milk = new RegularItem("Milk", 2);

list2.addItem(milk);

assert(list2.totalQuantity() == 2);

// Test Case 3: Multiple item types

GroceryList list3;

Item\* bread = new RegularItem("Bread", 2);

Item\* rice = new BulkItem("Rice", 5);

list3.addItem(bread);

list3.addItem(rice);

assert(list3.totalQuantity() == 7);

// Test Case 4: Edge case with zero or negative quantities

GroceryList list4;

Item\* eggs = new RegularItem("Eggs", 0);

Item\* flour = new BulkItem("Flour", -1);

list4.addItem(eggs);

list4.addItem(flour);

}

1. Create one integration test for one of the test cases.

int main() {

GroceryList myList;

Item\* milk = new RegularItem("Milk", 2);

Item\* rice = new BulkItem("Rice", 5);

myList.addItem(milk);

myList.addItem(rice);

int total = myList.totalQuantity();

assert(total == 7);

std::cout << "Integration test passed: totalQuantity() = " << total << std::endl;

return 0;

}

# Problem 3.6.2: Account Reports



For the above UML class diagram, please do the following:

1. Identify three methods which could benefit from an integration test. List them in order from the most beneficial to the least.

The three methods that would benefit most from integration testing are computeCategory(Filter), initialize(), and display(). The computeCategory(Filter) method should be tested first because it integrates data from multiple classes, including Report, Account, Category, and Filter, making it the most complex interaction. The initialize() method is next since it sets up relationships between reports, accounts, and categories, ensuring the system starts in a consistent state. Finally, the display() method benefits from integration testing because it relies on data generated by both initialization and computation to correctly output report information.

1. Create four test cases for one of the methods.

void test\_computeCategory() {

// Test Case 1: Empty report with no accounts

Report report1;

Filter filter1;

report1.computeCategory(filter1);

assert(report1.getSubtotals().empty());

// Test Case 2: Single account matching filter

Report report2;

Account acc1("Salary", 5000);

report2.addAccount(acc1);

Filter filter2("Salary");

report2.computeCategory(filter2);

assert(report2.getSubtotals()["Salary"] == 5000);

// Test Case 3: Multiple accounts with mixed filter matches

Report report3;

report3.addAccount(Account("Salary", 5000));

report3.addAccount(Account("Rent", -1500));

Filter filter3("Salary");

report3.computeCategory(filter3);

assert(report3.getSubtotals().size() == 1);

assert(report3.getSubtotals()["Salary"] == 5000);

// Test Case 4: Multiple accounts all matching filter

Report report4;

report4.addAccount(Account("Food", -200));

report4.addAccount(Account("Food", -300));

Filter filter4("Food");

report4.computeCategory(filter4);

assert(report4.getSubtotals()["Food"] == -500);

}

1. Create one integration test for one of the test cases.

int main() {

Report report;

report.addAccount(Account("Salary", 5000));

report.addAccount(Account("Rent", -1500));

Filter salaryFilter("Salary");

report.computeCategory(salaryFilter);

int subtotal = report.getSubtotals()["Salary"];

assert(subtotal == 5000);

std::cout << "Integration test passed: computeCategory() correctly filtered and summed 'Salary' category." << std::endl;

return 0;

}

# Problem 3.6.3: User Access



For the above UML class diagram, please do the following:

1. Identify three methods which could benefit from an integration test. List them in order from the most beneficial to the least.

The three methods that would benefit most from integration testing are canRead(Context), canWrite(Context), and initialize(). The canRead(Context) method should be tested first because it integrates the user’s access permissions with the given context and determines whether a read operation is allowed, involving both User and Access classes. The canWrite(Context) method is next, as it tests the same type of interaction but for write privileges, ensuring that write access is properly restricted based on user roles such as Administrator, Auditor, or Standard. Finally, the initialize() method benefits from integration testing because it sets up the user’s access rights and ensures each subclass properly configures its permissions according to its type.

1. Create four test cases for one of the methods.

void test\_canRead() {

// Test Case 1: Administrator has read access to all contexts

Administrator admin;

admin.initialize();

Context systemContext("System");

assert(admin.canRead(systemContext) == true);

// Test Case 2: Auditor has read access but not write access

Auditor auditor;

auditor.initialize();

Context auditContext("Logs");

assert(auditor.canRead(auditContext) == true);

// Test Case 3: Standard user can only read limited contexts

Standard standard;

standard.initialize();

Context publicContext("Public");

assert(standard.canRead(publicContext) == true);

// Test Case 4: Standard user denied access to restricted context

Context adminContext("AdminPanel");

assert(standard.canRead(adminContext) == false);

}

1. Create one integration test for one of the test cases.

int main() {

Standard standardUser;

standardUser.initialize();

Context publicContext("Public");

bool canRead = standardUser.canRead(publicContext);

assert(canRead == true);

std::cout << "Integration test passed: Standard user can read 'Public' context." << std::endl;

return 0;

}

# Problem 3.6.4: Address Book



For the above UML class diagram, please do the following:

1. Identify three methods which could benefit from an integration test. List them in order from the most beneficial to the least.

The three methods that would benefit most from integration testing are insert(), update(), and find(). The insert() method is the most important to test because it integrates the AddressBook class with the Address and Contact subclasses such as Phone, Email, and SocialMedia, ensuring new contact entries are added correctly. The update() method is next because it must correctly modify contact information across different subclasses and verify that the updated data is reflected consistently throughout the address book. Finally, the find() method benefits from integration testing because it checks the interaction between the search functionality and all stored Contact types, confirming that the system retrieves the correct contact information regardless of the contact type.

1. Create four test cases for one of the methods.

void test\_insert() {

// Test Case 1: Insert a phone contact

AddressBook book1;

Contact\* phone = new Phone("555-1234");

book1.insert(phone);

assert(book1.find("555-1234") == true);

// Test Case 2: Insert an email contact

AddressBook book2;

Contact\* email = new Email("user@example.com");

book2.insert(email);

assert(book2.find("user@example.com") == true);

// Test Case 3: Insert a social media contact

AddressBook book3;

Contact\* social = new SocialMedia("username123", "password");

book3.insert(social);

assert(book3.find("username123") == true);

// Test Case 4: Attempt to insert a duplicate contact

AddressBook book4;

Contact\* dupEmail = new Email("duplicate@example.com");

book4.insert(dupEmail);

book4.insert(dupEmail); // Duplicate

assert(book4.count("duplicate@example.com") == 1);

}

1. Create one integration test for one of the test cases.

int main() {

AddressBook book;

Contact\* social = new SocialMedia("user456", "password123");

book.insert(social);

bool found = book.find("user456");

assert(found == true);

std::cout << "Integration test passed: SocialMedia contact successfully inserted and found in AddressBook." << std::endl;

return 0;

}