Subject: Software testing and TDD

The big picture: We're developing an inventory and purchasing system for a regional franchise, our client. The complete software system will include an inventory management system and will control the point-of-sale checkout registers. Our team is responsible for the point-of-sale software, not the inventory system. Because of the financial, tax, and legal liability of the client, the software accuracy must be ensured and verified.

User Story: Compute the net price of all the items in a customer's shopping cart including all applicable discounts and taxes.
After discussing further with the client, we have defined the following *functional requirements*.

FR 1) If the shopping cart contains 10 or more items, the customers shall get a 10% discount off the purchase price before taxes are included. If the shopping cart contains more than 5 items, the customer shall receive a 5% discount before tax. Otherwise, no discount is applied.

FR 2) Customers may not purchase more than 50 items in a single checkout session.

FR 3) If the customer is a member of the store's discount shopping club, they shall receive an additional 10% discount off the purchase price before taxes.

FR 4) Unless a customer has tax-exempt status, the local sales tax rate of 4.5% shall be applied to the net discounted price.

FR 5) Dollar amounts shall be rounded to the nearest cent ($0.01) before and after the tax rate is applied.

FR 6) The net purchase before any discount, the net discount, the tax amount, and the net total due shall all be reported in US dollars.

Part 1:
In Part 1, you task is to design test cases for the above requirements. No coding is required! Here are the specific tasks for this part.

a. Analyze the first four functional requirements above (i.e., FR 1-4) and determine the number of test cases needed to provide 100% path and conditional coverage. Use graph theory (i.e., control flow analysis) and / or conditional truth tables to analyze the requirements.
b. Design additional tests using equivalence partitions and boundary value analysis (BVA) to provide further verification for FR 1 and FR 2.
c. Design tests for FR 5 using EP and BVA. Pay attention to the float-point nature of the values!
d. Create a document enumerating each test case and describe the intention of the test and the expected outcome. Spreadsheets are useful for this task but another option is to use Markdown (or similar source documentation language.) The benefit of the Markdown approach is that it can be managed like any other bit source code.

Submit your analysis of the theoretical number of test cases to Sakai. Be sure to describe how you deduced the number of test cases for each requirement and provide documentation (e.g., flow graphs, pseudo-code, truth tables) and justification for your design.

Part 2:
In Part 2, you'll implement the above user story using an object-oriented (OO) language of your choosing. Implement the functionality incrementally following the test-drive development (TDD) principle (as much as possible). Also, implement the software with
Here are the specific tasks for Part 2.

a. Create automated tests cases that provide statement, branch, conditional, and loop coverage as necessary. Create additional test cases using equivalence partitioning (EP) and boundary value analysis (BVA) techniques as needed. Note, that you have already designed the tests in Part 1 but verify that you still have adequate coverage.
b. Keep a test log documenting what bugs were found by which test case and why the test failed.
c. When implementing the test cases, include source-code documentation (i.e., in-lined in the source) describing the intent and motivation for each test.
d. Create mock objects as needed to complete the implementation and testing.
e. Use git version control and publish to BitBicket. Each increment should have at least one commit and the commit history should track the evolution of the TDD process.

Development details:

The larger inventory database system is not complete, but the software design of the database defines a `DataBase` class that has a `getItem` member function:

```
Item DataBase::getItem (String ItemID)
```

that returns a pointer to an object of type `Item` if found; otherwise, it returns NULL. (Translate this behavior to whatever language you select. For example: in Python, the method will return a reference to an object of type `Item` if found; otherwise, it returns `None`.)

The `Item` class has a `getPrice` member function:

```
float Item::getPrice()
```

that returns the purchase price of a single item.

Implement the method

```
ShoppingCast::calculatePurchasePrice (
          List<String> ProductIDs, Customer customer)
```

which is a member of the `ShoppingCart` class. `ProductIDs` is a list (i.e., sequence) of strings (in your favorite OO language) that represent the item's unique identifier of all the products in the customer's shopping cart (assume they were scanned already) and `Customer` is a class with methods indicating if the customer is a member of the store's Discount Shoppers Club and if they are tax-exempt. I leave it to you and your partner (if you choose) to design of the `Customer` class and the list of strings parameters passed to the `calculatePurchasePrice` method.

You will need to create a mock database instantiation of type `DataBase` to supply information to the `ShoppingCart` class method(s). The mock object can be instantiated and passed to the constructor of your `ShoppingCart` to avoid inheritance issues when the real data type is included.

Commit your OO source code and automated test cases to BitBucket and test log to Sakai. **The submitted code must be professional quality.**

Include enough documentation to allow an outside developer or tester to create and execute black-box test cases based upon the specified user requirements.

Include instructions for building and linking your implementation with the test driver.

You should separate the class signatures from the implementations as much as possible within the confines of your chosen programming language. For example, separate class headers (.h, .hpp) and implementation files (.cpp) in C++.

Advice: It is good practice to avoid publishing source or build methods that are dependent upon a specific IDE. Ideally, your code should be structured so that it can be built without an IDE; however, this is not a requirement for this assignment. Why is this an issue? When publishing (or delivering) executable software systems, the build environment is not particularly relevant. However, when publishing source code, the intent is that other developers can build your suite. Tight dependency upon a specific IDE can be a burden and anything that is burdensome reduces uptake.