QAP 1 – SD12 Semester four

Due Date: June 1st, 2025

Author: Noah Devine

1. **Screenshots & Clean Code:**

```java
@Test
void testMakePayment() {
    testCard.chargeToCard(new Money(amount:400.00));
    testCard.makePayment(new Money(amount:150.00));
    assertEquals(expected:"$250.00", testCard.getBalance().toString());
}
```

Example 1: Meaningful Names

The above screenshot shows an example of clean code by using clear, descriptive naming conventions. Users/devs should be able to read the names of your functions/methods and get a clear understanding of the core functionality of what your code is doing. testMakePayment() clearly indicates that the test is verifying that the makePayment() method is making a payment on a card.

Example 2: Clean Unit Tests

As outlined in the O'Reilly book, unit tests should have a single concept per test. This means avoiding long, unfocused test methods that try to verify multiple behaviors at once. The above screenshot also applies to this. In the above test, we are only testing the one thing which is making the payment on the credit card and not testing other logic such as charging the credit card or the actual card initialization. This test is also fast, and independent from the other test, which follows the best clean coding practices for unit testing.

```java
/**
 * A class for handling money in dollars and cents
 * This class allows you to add, subtract, and compare amounts without worrying about rounding issues
 * Stores dollars and cents separately.
 */
public class Money {
    private final long dollars;      You, 2 days ago • General code cleanup …
    private final long cents;
```

Example 3: Clean Comments

The above comment demonstrates the use of good comments for clean coding best practices. The above comment avoids unnecessary repetition and focuses on the intent behind the class, which is handling money. Instead of just restating what the code does, this comment explains design choices like separating dollars and cents to avoid rounding issues.

2. Project Explanation

This project is a simple simulation of a banking credit card system. This program allows you to create a person with an address and then assign the credit card to them. The program also allows you to simulate real world credit card uses like making payments and charging the credit card. This project includes four classes:

1. Address Class – Stores basic location info like street, city, state, and ZIP code.
2. Person Class – Represents someone with a name and a home address.
3. Money Class – Handles dollar and cent values.
4. CreditCard Class – Lets a user own a credit card, make charges and pay it off while making sure they don't go over their credit limit.

I've also created a demo class for testing purposes so you can see the project in action.

Test Cases:

AddressTest – Tests initialization of an address and the toString method.

PersonTest – Tests initialization of a person object and behavior when the address is Null.

MoneyTest – Tests correct results from the add and subtract methods while also testing proper comparison using IsEqualTo.

CreditCardTest – Tests credit card initialization, making payments, charging to the card within its credit limit, and charging over the credit limit.

3. Needed Dependencies

- Junit5 for the unit tests
- Maven for project management

Acquired from Maven Central

4. **QAP issues:**
    - Had some issues setting up the project base structure but that might have been a skill issue on my part.
    - IntelliJ was giving me grief for a moment but again, skill issue in my part. I still require more practice with that IDE.