Title: Design Assessment
Authors: Andrew Fulton and Jazz Loffredo

Class: CIS501, Fall 2018

1)

```
public class Fruit
{
    String name;
    int color;
    int shape;
    Genus genus;

    public Fruit(String name, int emp_id, Genus genus)
    {
        // Update the data member;
    }

    int UpdateShape(int salary)
    {
        // Update shape of Fruit
        // ..............
        // ........
        return 0;
    }

    void UpdateGenus(Genus genus)
    {
        // Change genus of Fruit
        // ...code goes here
        // ......
    }
}

public class FruitRepository
{
    private DatabaseHandler _db;

    public FruitRepository(DatabaseHandler db)
    {
    }

    public int Update(Fruit fruit)
    {
        // Update shape of Fruit in database
```

```
        // ..............
        // ........
        return 0;
    }
}
```

Problems and Solution: The comment at the top of the file states, “The purpose of the class is to maintain the correct state of Fruit objects.” We noticed that the fruit class was attempting to handle various operations, therefore breaking the single responsibility principle. We decided that the handling of the database interaction should occur in a separate class titled FruitRepository. Within this class, there is a private variable of type DatabaseHandler. We moved the update_Fruit_database(DatabaseHandler _db) into our new FruitRepository class. This new class is now solely responsible for updating Fruit in the database while our Fruit class is solely responsible for managing information about the current state of a Fruit object. We are now within the bounds of the single-responsibility-principle such that each class only has one responsibility.

2)

```
public interface RebateAccount
{
    double getRebate(double totalSales);
}

public interface StarAccount
{
    void AddStarPoints(int points);
}

public abstract class RebateClient : RebateAccount
{
    protected RebateClient(double rebatePercentage)
    {
        RebatePercentage = rebatePercentage;
    }

    protected double RebatePercentage { get; }

    public double getRebate(double totalSales) =>
        totalSales - (totalSales * RebatePercentage);
}

public abstract class Client : RebateClient, StarAccount
```

```
{
    protected Client(string clientType, double rebatePercentage)
        : base(rebatePercentage)
    {
        ClientType = clientType;
    }

    protected string ClientType { get; }

    public void AddStarPoints(int points) =>
        Console.WriteLine($"Adding {points} points to {ClientType} Client's Star account");
}

public class PotentialClient : RebateClient
{
    public PotentialClient()
        : base(.1)
    {
    }
}

public class GoldClient : Client
{
    public GoldClient()
        : base("Gold", .25)
    {
    }
}

public class PlatinumClient : Client
{
    public PlatinumClient()
        : base("Platinum", .5)
    {
    }
}
```

Problems and Solution: Upon initial inspection of this set of classes, we noticed that each variation of client was using inheritance as a form of implementation. In the Client class, there are two methods defined and implemented: GetRebate(double totalSales) and AddStarPoints(int points). GoldClient, PlatinumClient, and PotentialClient all override these methods with their own implementations. According to the Liskov-Substitution Principle, we could not swap a generic Client for a GoldClient/PlatinumClient/PotentialClient since each of their implementations are different. This is a

case where our parent is a real duck and our subclasses are rubber ducks. We cannot substitute the child classes for our parent class. Our proposed solution involves creating two interfaces: RebateAccount and StarAccount, each containing method definitions for the methods that will be overridden by the various clients. We also updated Client to be an abstract class so that Platinum/Gold clients would all extend this generic Client class with their own implementations of the above interfaces. RebateClient is its own abstract class that only implements the RebateAccount interface. The Potential client extends the RebateClient since, as stated in the original file, Potential clients do not have a star account, but they are given a rebate.

3)

```
public interface IEnrollment
{
    double GetDiscount(double tuition);
}

public class EnrollmentTypeOne : IEnrollment
{
    public double GetDiscount(double tuition)
    {
        return tuition - 100;
    }
}

public class EnrollmentTypeOther : IEnrollment
{
    public double GetDiscount(double tuition)
    {
        return tuition - 50;
    }
}
```

Problems and Solution: Upon initial inspection of the Enrollment class, we noticed that there is a private integer variable called _EnrollmentType. This was then used in the GetDiscount(double Tuition) method to essentially override the method depending on the EnrollmentType (two different implementations of GetDiscount() contained within the method). This is similar to the example slides in which we had multiple potential classes with varying implementations of functions. Our proposed solution involves creating an interface Enrollment class which is extended by EnrollmentTypeOne and EnrollmentTypeOther. The EnrollmentTypeOne class overrides the GetDiscount(double Tuition) method so that it is equivalent to the original class that checks for if(EnrollmentType == 1). The EnrollmentTypeOther class overrides the GetDiscount() method and implements the else portion in the

original class. With this new implementation, we can extend the functionality of EnrollmentTypeOne without affecting the EnrollmentTypeOther class. With the original code, for each extension of functionality, we would have to check the integer _EnrollmentType first. With our proposed solution, we no longer have to recompile, say EnrollmentTypeOne, if we make changes to EnrollmentTypeOther. This allows our code to be open to extension while still being closed to modification. We can also have a list of Enrollment objects and we do not have to type check, we can simply call the GetDiscount(double Tuition) method and it will automatically invoke the correct overridden method.