Designing for testability

1. When is a class controllable? How do you make a class controllable?

Controllable means easily control what the class under test does. One way is by making the dependencies easily replaceable by a test double (including injecting the test double via for example a constructor).

2. When is a class observable? How do you make a class observable?

Observable means you can see what is going on with the class under test and inspect its outcome. Couple ways are by introducing methods to facilitate assertions and changing void method to return.

3. The book says: *whenever we need a spy to assert the behavior, we must ask ourselves why we need a spy*. Why is this?

There may be other (easier) ways to test the right behavior for example a True/False method like `isReadyForDelivery` without a spy.

4. Is the change shown in Listing 7.11 in the book a good solution to improve observability?

The change is from a void function to a function that returns a list with the objects that it has created. It's probably a good change, but not necessarily (depends on the application).

5. Is this testable? If not, why not?

```
1  class InvoiceFilter {
2     public List<Invoice> lowValueInvoices() {
3         DatabaseConnection dbConnection = new DatabaseConnection();
4         IssuedInvoices issuedInvoices = new IssuedInvoices(dbConnection);
5
6         List<Invoice> all = issuedInvoices.all();
7         return all.stream()
8                 .filter(invoice -> invoice.getValue() < 100)
9                 .collect(toList());
10    }
11 }
```

6. Would you create a Clock class and create an attribute of type Clock in ChristmasDiscount? Or pass a value of type LocalDate to the applyDiscount method? Why?

```
public class ChristmasDiscount {
   public double applyDiscount(double amount) {
       LocalDate today = LocalDate.now();
       double discountPercentage = 0;
       boolean isChristmas = today.getMonth() == Month.DECEMBER
               && today.getDayOfMonth() == 25;
       if (isChristmas)
           discountPercentage = 0.15;
       return amount - (amount * discountPercentage);
   }
}
```

Create a Clock class because it does not change the usage of the applyDiscount method. (although changing the method would make it easier for testability)

7. How can you improve the testability of the following OrderDeliveryBatch class?

```
public class OrderDeliveryBatch {
    public void runBatch() {
        OrderDao dao = new OrderDao();
        DeliveryStartProcess delivery = new DeliveryStartProcess();
        List<Order> orders = dao.paidButNotDelivered();
        for (Order order : orders) {
          delivery.start(order);
          if (order.isInternational()) {
            order.setDeliveryDate("5 days from now");
          } else {
            order.setDeliveryDate("2 days from now");
          }
       }
    }
}
class OrderDao {
  // accesses a database
}
class DeliveryStartProcess {
  // communicates with a third-party web service
}
```

To test the runBatch method of OrderDeliveryBatch (for example, in a unit test), you need to be able to use mocks or stubs for at least the orderBook and delivery objects. In the current implementation, this is not possible, as you cannot change orderBook or delivery from outside the class. In other words, you want to improve controllability to improve testability. If you allow the dependencies to be injected, you will be able to use mocks and stubs. Therefore, you should consider dependency injection.

8. How can you improve the testability of the following KingsDayDiscount class?

```
1 public class KingsDayDiscount {
2   public double discount(double value) {
3     Calendar today = Calendar.getInstance();
4     boolean isKingsDay = today.get(MONTH) == Calendar.APRIL
5                               && today.get(DAY_OF_MONTH) == 27;
6     return isKingsDay ? value * 0.15 : 0;
7   }
8 }
```

The implementation currently lacks controllability. You cannot change the values that Calendar gives in the method because its getInstance method is static. Although newer versions of Mockito can mock static methods, try to avoid them as much as possible. A solution would be to either receive Calendar as a parameter of the method or inject a Clock, which is a layer on top of Calendar (or whatever other Date class you prefer)