Jon Doretti

SE333

4/27/2022

1.a)

I) Order.java; Duplicate code – calculate total line items is implemented in the code many times.

II) Order.java; Message chain – sql = new StringBuffer().append().append().append().append().toString()

III) LineItemList.java; Data Class/Lazy Class – LineitemList.java only has one getter and setter.

1.b) To solve the issue of duplicate code in order.java, the programmer can create a method for calculating the total for line items. That way they can call the method to execute the same four lines of code. For the message chain antipattern found also in Order.java, the programmer can use one append statement and include all Strings inside of that append. Also since it calls .toString at the end, I believe the developer can just hard code the string into String sql because .toString() returns a string representing the data in the sequence. Lastly for the lazy class – the getter and setter can be implemented into Oder.java or left inside of lineItem.java itself.

2) The first design antipattern is duplicated code. This appears in Reservation and Books both have getBookID and setBookID(). These are duplicate expressions in different classes. The second design antipattern is the large method inside of Patron. The number of methods per class is twenty in the Patron class. Patron is doing a lot that can be done in other classes for example getFine: Double and setFine(Double) should be done in the Fine class not the Patron.

3) Refactoring is changing the structure of the classes, methods, and structure of the code without changing the behavior of the code. For the duplicated code antipattern getBookID and setBookID() should be handled by the Books class and then and instance of a book should be created in Reservation where they can call the methods when needed. Also inside Books getBookID(), it should change to take an int; getBookID(): Int and same for setBookID(Int): void. For the large method antipattern inside of the Patron class, Patron should be refactored to split some of the responsibilities and create separate methods. For example, instead of collecting the contact information all inside Patron with getName():String, getAddress(): String, getPhone(): String and getEmail(): String with associated setters, there should be a contact information class as if a new class wants to be introduced for employees all that information could be used there too. This would help promote re-usability of a class and higher levels of abstraction.

4) A defect is a problem with the code that is not intentionally implemented into the code. A defect can occur after many iterations of a code base or during the first implementation. An antipattern is poor

coding and design choices that effect many aspects of the software, for example, reusability and understandability.

5.1) Testing is to show that there are no errors/bugs/defects in the software: True

5.2) Applying refactoring cleans up the software from bug/bugs: False

Justification: While refactoring 'could' fix a bug – refactoring works to help restructure the code to remove code smell/antipatterns while not changing the external behavior.

5.3) Coupling generally refers to relationship between elements belonging to the same module, whereas cohesion refers to relationship between elements in different modules: False

Justification: Cohesion refers to the relationship between elements belonging to the same module, and coupling refers to the relationship between elements in different modules.

5.4) The existence of errors in the modules does not necessarily imply the existence of faults in the system: True

5.5) There is no consensus about how to define and measure software design defects (codesmells): True

5.6) A system may deliver the expected result even with the existence of faults: True

5.7) A high value of Depth Inheritance Tree (DIT) is an indicator of a healthy code: False

Justification: The deeper the tree the greater the design complexity; hence, smaller values are better.

5.8) An example of refactoring is adjusting of existing features to satisfy a customer requirement discovered after a project is shipped: False

Justification: The definition of refactoring calls for the external behavior to not be changed; therefore, adjusting an existing feature to satisfy a new requirement is not refactoring.

5.9) 9 Defects are synonymous with faults: True

6.a)

```
Class UserAccount{

        private String userID;

        private String password;

        private String[] securityQuestion;
        private String[] securityResponce;
        private Date dateCreated;
        private Date dateLastModified;
        private Date dateClosed;
        AccountHolder acctHolder;
}

class AccountHolder{
        protected Address address;
        protected Phone phone;
        protected String email;
        List<BankAccount> bankAcct;
        UserAccount userAcct;
}

class Merchant extends AccountHolder{
        private String name;
        private Name contact
}

class Customer extends AccountHolder{
        private Name name;
        private String[] authenticationDate;
}

class Transfer{
        private Date dateTransferred;
        private Currency amount;
        List<BankAccount> bankAccount;
        List<AccountHolder> acctHolder;
}
class BankAccount{
        protected String accountNo;
        protected Currency balance;
        protected String status;
        protected Date dateOpened;
        protected Date dateClosed;
        Transfer transfer;
        List<AccountHolder> acctHolder;
}
class BusCheckingAcct extends BankAccount{
        private String busAcctType;
        private String terms;
}

class OtherAccount extends BankAccount{
        private String type;
}

class  CheckingAcct extends BankAccount{
        private Currency minBalance;
        private Currency overdraftFee;
}
```

6.b)

```
class D {
        List<A> a;
}
```

```
class A {
        D d;
}
class B extends A {
}
class C extends A {


}
```