HW5 - Answers

Answer 1:

i) In the example provided, the bad code is in the method “price()”. There are a few issues with the way it has been coded.

a) The operation “_quantity * _itemPrice” is performed twice within the same method
b) Both the “Quantity Discount” and the “Shipping Charge” is calculated within the same method

It would be cleaner to refactor this code and moving the individual operations into their own methods. This way it can be reused elsewhere too. We can go ahead and do this by following the concept of “Extract” method.

Step 1: We can move the logic for calculating the Base Price into its own method

```
private double calculateBasePrice(){
    return _quantity * _itemPrice;
}
```

Step 2: Now move the logic to calculate the Quantity Discount and Shipping Charge to their own methods too.

```
private double calculateQuantityDiscount(){
   // quantity discount
   return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
}

private double calculateShipping(){
    // shipping
    return Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Step 3: Now we can replace the logic within the price() method internally with the 3 new method calls.

```
public double price(){
  return calculateBasePrice()  - calculateQuantityDiscount() + calculateShipping()
}
```

After this, the refactored code would finally look like this

```
public class Order {

        private String _customer;
        private int _quantity;
        private double _itemPrice;
```

```
    public Order (String customer) {
        _customer = customer;
    }

    public String getCustomer() {
        return _customer;
    }

    public void setCustomer(String customer) {
        _customer = customer;
    }

    public double price(){
       //price is base price - quantity discount + shipping
       return calculateBasePrice()  - calculateQuantityDiscount() + calculateShipping()
    }

    private double calculateBasePrice(){
      // base price
      return _quantity * _itemPrice;
    }

    private double calculateQuantityDiscount(){
      // quantity discount
      return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
     }

     private double calculateShipping(){
      // shipping
      return Math.min(_quantity * _itemPrice * 0.1, 100.0);
    }
}
```

ii) In the example provided, each Order object has its own Customer object and is treated as a value object. To add details like Address or Credit Rating to a Customer object, we will have to use the concept of “Change Value to Reference”.

The idea behind this is to modify the code such that all orders for the same customer share a single Customer object. The One to One relationship that exists currently between the Order and the Customer object must be replaced with a Many to One Relationship such that any number of orders can share a single Customer object.

We can achieve this in the following fashion:

Step 1: Replace the constructor of the Customer object with a Factory Method

```
class Customer{
```

```
    //Make the Constructor private
    private Customer(String name){
          _name = name;
    }

    //Define the Factory Method which will be used to return the Customer object
    public static Customer create(String name){
          return new Customer(name);
    }
}
```

Step 2: Modify the call to the Customer object in the Order class

```
class Order{

    public Order(String customer){
        _customer = Customer.create(customer);
     }

}
```

Step 3: Now we have to decide how to access the Customers. This can be done, in this case, by making the Customer class as an access point. The customers could be created on the fly when required or preloaded. If we decide to go with the pre-load option, then we can load the customers that are in use by retrieving either from a DB or a File

```
class Customer{

   static void loadCustomers(){
     new Customer("A").store();
     new Customer("B").store();
   }

}
```

Step 4: We can now modify the create method and rename it to indicate that it only returns an existing customer.

```
class Customer{

    public static Customer getNamed(String name){
      return (Customer) _instances.get(name);
    }

}
```

The Customer object can contain any number of attributes that the store desires such as the Address, Credit Rating etc.,

Answer 2:

It is obvious from the code that the Person class is handling more than its fair share of responsibilities. For example, the details "_officeAreaCode", "_officeNumber" are generic details and are not specific to a particular user, even though a specific telephone number might have been assigned to an user. The class has data which should be handled by two classes at least, in this case.

We can fix the problem with this class by splitting up the responsibilities into two classes by using the refactoring principle "Extract Class". As per this, when we have one class doing work that should be done by two, we create a new class and move the relevant fields and methods from the old class into the new class.

The steps involved are as given below:

i) Create a new class specifically for handling the Telephone Number functionality
ii) Move the attributes _areaCode and _telephoneNumber to the new class using the "Move Field" refactoring methodology
iii) Move the Methods from the old class to the new one using the "Move Method" refactoring.
iv) Make sure to compile and test the code after every move.
v) Review the code to ensure that it still performs the desired operations

The newly modified code should look like this now with a clearly defined functionality.

```
public class TelephoneNumber{

        //Add a method to return the constructed phone number
        public String getTelephoneNumber(){
                return ("(" + _areaCode + ")" + _number;
        }

        private String _number;
        private String _areaCode;

        //Add the Setters and Getters to access the private variables
        public String getNumber(){
                return _number;
        }
        public void setNumber(String p_number){
                _number = p_number;
        }

        public String getAreaCode(){
                return _areaCode;
        }
        public void setAreaCode(String p_areaCode){
                _areaCode = p_areaCode;
        }
}
```

vi) The initial Person class can now be cleaned up to take advantage of the new TelephoneNumber class. We should also rename the original class if its responsibilities do not match its name anymore to make it more legible.

```
class Person {

   private String _name;
   private TelephoneNumber _officeTelephoneNumber = new TelephoneNumber();

   public String getName() {
      return _name;
   }

   public String getTelephoneNumber() {
     //Use the newly created class and its method to retrieve the Telephone Number for a specific
person
      return _officeTelephoneNumber.getTelephoneNumber();
   }

}
```

Answer 3:

The relationship defined in the question between the Customer and the Order class implies a Unidirectional Many to One relation between the Order and the Customer. In order to allow a Customer to have many orders and also to share an order among different customers, we will have to modify this relationship so that it is Bidirectional Many to One between the Order and the Customer.

Using the refactoring concept of “Change Unidirectional Association to Bidirectional”, we can perform this operation using the following steps

i) In the example provided, the Customer class has no reference to the Order class. We can start by refactoring this by adding a field to the Customer class. Since the Customer can have multiple orders, we need a collection. However since the orders can’t be repeated for the same customer, we can use the Set collection to maintain the uniqueness.

```
class Customer {

  private String _name;
  private String _addr;
  private int _age;
   …

   //Add a Set of Orders to the Customer class since a Customer can have multiple orders
   private Set _orders = new HashSet();
}
```

ii) We have to decide which way the association holds and which class takes charge of the association. In this example, the Order class takes charge since it has a Many to One relationship with the Customer class. The Order class basically controls the association. We need to add a helper method in order to allow the customer to access the orders collection.

We can do this by adding a method which will help us serve this purpose.

```
class Customer{

  Set friendOrders(){
     //This will be used to return the Orders set
     return _orders;
  }

}
```

iii) Update the modifiers in the Customer class to update the back pointers

```
   class Order{

      void setCustomer(Customer customer){
        if(_customer != null)
              _customer.friendOrders().remove(this);
         _customer = customer;
         if(_customer != null)
             _customer.friendOrders().add(this);
      }

 }
```

iv) Now we can modify the link in the Customer class so that it can call the Order class. However since an Order can have many Customers, it becomes a Many to Many relationship between the Order and the Customer.

The Customer class can be modified to add or remove orders as follows:

```
  class Customer{

        void addOrder(Order order){
           order.addCustomer(this);
        }


        void removeOrder(Order order){
            order.removeCustomer(this);
        }

}
```

The Order class will look like this, in order to be able to Add or Remove Orders.

```
 class Order{

   void addCustomer(Customer customer){
     customer.friendOrders().add(this);
     _customers.add(customer);
    }

   void removeCustomer(Customer customer){
     customer.friendOrders().remove(this);
     _customers.remove(customer);
   }
}
```

Thus, by establishing this relationship we can ensure that the states between the Order class and the Customer class remain consistent too.

Answer 4:

Taxonomies	**Code Smell**
Bloaters	Data Clumps Large Class Long Method Long Parameter List Primitive Obsession
OO Abusers	Alternative Classes with Different Interfaces Refused Bequest Switch Statements Temporary Field
Change Preventers	Divergent Change Parallel Inheritance Hierarchies Shotgun Surgery
Dispensables	Data class Duplicate Code Lazy class Speculative Generality Dead Code
Couplers	Feature Envy Inappropriate Intimacy Message Chains Middle Man

Answer 5:

Refactoring is essential in the field of Software engineering.

A few reasons why refactoring is needed can be classified as follows:

i) To improve the design of software: A code is continually changed to add or remove functionalities. If considerable thought is not given during these changes, we usually end up with a structureless code. The design of the program also begins to decay if not refactored. Refactoring is akin to tidying up a code. It helps us remove bits of code that aren't really needed anymore.One of the other important aspect of improving design by refactoring is to weed out duplicate code. Reducing the amount of code makes a big difference in modification of the code. By eliminating duplicate code, we can ensure that the code is legible and its design is obvious to any developer, which is the essence of any good design code.

ii) Makes software easier to understand: Refactoring helps in making a software easier to understand, making it easy to maintain the code for any developer. If due consideration is not given during coding, it can cause maintenance problems later on for future developers. Refactoring helps to make the code more readable. Similarly, it helps one to understand unfamiliar code also, because by using refactoring, we can comprehend the code better and get a better grasp of the intended design. Refactoring leads to higher levels of understanding which would be otherwise missed due to a badly structured code.

iii) Identify bugs: It helps to identify bugs which might be missed in a clump of code. By refactoring we can understand the structure of the code better and thereby easily spot bugs in the existing code.

iv) Program Faster: As per Fowler, Refactoring also helps to develop code more quickly. A good code which is constantly refactored is cleaner and easier to understand. It improves the quality of the code. Improving design, improving readability, reducing bugs, all these help in definitely improving the quality of the software. The speed of development is increased because less time has to be spent on understanding the code or finding and fixing bugs. A well designed code requires less patching and is essential in maintaining speed in software development.

In order to identify when it is appropriate to Refactor, there are three stages identified when it can be done most efficiently

i) Add a Function: One of the best times to refactor a code is whenever we add a function. To implement a function, we need to spend time on understanding the code first. This serves as a good reason for refactoring the code to get a better understanding. So if it is easier to understand the code by refactoring, then we should go ahead and do it, while adding a function. Similarly, if the design of the code makes it difficult to add new features then it is a good time to refactor the code to make it more understandable

ii) Fix a Bug: While trying to fix bugs, we need to make the code more understandable in order to do it. So if we refactor the code during this process, it makes it easier to understand the code and helps in fixing the bug quicker and better. The other way of looking at it is, if a bug

is found in a code then it's a sign to consider refactoring. The idea being, if the code was clear in the first place, then the bug would not exist at all.

iii) Code Review: Refactoring helps in reviewing other's code too. If a suggestion is provided as part of refactoring, then it is usually implemented immediately, when done as part of the code review process. It is also very important in writing a clear code. A person writing code might have their own way of looking at things which might not necessarily be correct. This may result in unstructured code and bad design. During code review process, different people look at the code and provide suggestions to improve the code. So any refactoring done to improve the code makes that task much easier for the review team. It is definitely one of the best times to refractor a code.

Answer 6:

One of the examples of refactoring that can't be completely automated is the Refactoring of Databases. Most of the systems are tightly coupled to a database schema that supports them. This is one of the reasons it is difficult to change them. Also a change in database schema forces us to migrate the data which can be a difficult task, even if we have layered the system to minimize the dependencies. These dependencies are something which can't be refactored as easily as it can be done for other Objects.

The other example could be the changing of Interfaces. Objects allow us to change the implementation of the software module separately from an interface. However if an interface is changed, then it has a ripple effect. This especially holds true for a published interface. Once it is published, it can not longer be "safely" changed. No IDE plugin would be able to decipher the various usage of the initial published interface and provide a safe way to refactor it.

Answer 7:

Based on the references given, the two features that are provided by IntelliJ but not by Eclipse are:

i) Type Migration: IntelliJ takes care of automatically applying the changes made to method return types, local variables, parameters and other data-flow-dependent type entries across the entire project. It also allows to switch between arrays and collections. IntelliJ takes care of making the changes for you.

ii) String Fragments: IntelliJ provides option to extract a part of a string expression. This feature is not present in other IDEs. If we select the fragment we need, IntelliJ takes care of the rest.

The other unique features listed for IntelliJ IDE are:

i) Make static
ii) Inline super class
iii) Replace inheritance with delegation
iv) Extract method object
v) Remove middleman
vi) Wrap return value

vii) Move instance method
viii) Convert to instance method
ix) Replace temp with query