Tutorial Sheet: Refactoring and Code Smells

Part A

1. **Which of the following best describes “Refactoring”?**
   - ▢ Translating code from one language to another
   - ▢ Changing external system behavior for performance gains
   - ▢ Improving internal structure without altering external behavior
   - ▢ Rewriting entire classes from scratch to implement new features
2. **A “Long Method” code smell can often be remedied by:**
   - ▢ Inlining all code into a single main method
   - ▢ Extracting smaller helper methods to break the logic into manageable chunks
   - ▢ Replacing the method with a single comment describing the logic
   - ▢ Deleting half the code until it’s short enough
3. **Which refactoring technique is used to remove a giant `if` or `switch` statement that checks object types?**
   - ▢ Extract Method
   - ▢ Replace Conditional with Polymorphism
   - ▢ Inline Method
   - ▢ Encapsulate Field
4. **When code in *Class A* heavily uses data from *Class B* rather than its own fields, it indicates which code smell?**
   - ▢ Feature Envy
   - ▢ Primitive Obsession
   - ▢ Long Method
   - ▢ Divergent Change
5. **In the context of refactoring, the principle that each small transformation should preserve existing behavior is crucial because:**
   - ▢ It allows skipping unit tests
   - ▢ It ensures we can reliably test incremental changes without new bugs
   - ▢ It requires rewriting all classes at once
   - ▢ It eliminates the need for stakeholders to review changes

6. **A “God Class” or “Large Class” is considered a smell primarily because:**
    - ▢ It uses inheritance incorrectly
    - ▢ It violates the Law of Demeter
    - ▢ It lumps too many responsibilities, harming cohesion and maintainability
    - ▢ It relies on a 3rd-party library
7. **Which of the following code smells deals with too many parameters or too many data fields always used together?**
    - ▢ Middle Man
    - ▢ Data Clumps
    - ▢ Speculative Generality
    - ▢ Switch Statement
8. **What is a recommended approach to addressing a “Long Parameter List” smell?**
    - ▢ Convert them into a single giant array
    - ▢ Introduce a parameter object or extract objects that group related data
    - ▢ Use reflection to pass dynamic arguments
    - ▢ Always add more parameters to simplify logic
9. **Polymorphism is often used to**
    - ▢ Make code slower but more stable
    - ▢ Replace class hierarchies with big conditionals
    - ▢ Eliminate conditionals that branch based on type codes
    - ▢ Force a single method to handle all object types via a `switch`
10. **When a method calls `obj.getSomething().getPart().doIt()` repeatedly, it’s known as**
    - ▢ A Message Chain smell
    - ▢ A Magic Number smell
    - ▢ A Bloaters problem
    - ▢ A YOLO pattern

Part B

1. Describe **three** distinct code smells you have encountered (or might encounter) in a large OOP codebase. For each, explain *why* it's considered a smell and how it impacts maintainability.
2. Compare and contrast these two smells. In which scenarios would "long method" be an easier fix? In which scenarios might "large class" be more problematic?
3. You see repeated parameters `(x, y, width, height)` across many methods. Why is this a smell, and how would you fix it? Provide a conceptual or real example.
4. Explain how message chains hamper code clarity. Provide a small code snippet showing a chain and how you might refactor it.
5. Why is "feature envy" contradictory to the principle of good OO design? Provide a short example illustrating the problem and how you'd move or split responsibilities.
6. Differentiate between "refactoring" and "rewriting from scratch." Under which conditions might you prefer a rewrite? Under which conditions is refactoring more appropriate?

Part C

1. Assume there is a method that prints invoice details, calculates tax, updates a DB record, logs info, and returns a status code. Show how you would break it into smaller methods. Provide *before* and *after* code.
2. You have a method:

   ```
   public double shippingCost(String shippingType, double
   weight) {
    switch(shippingType) {
        case "AIR": return weight * 2.0;
        case "SEA": return weight * 1.0;
        case "LAND": return weight * 1.5;
        default: return 0.0;
    }
   }
   ```

Show how to replace it with polymorphism. Provide interface or abstract class `Shipping` and concrete classes `AirShipping`, `SeaShipping`, etc.
3. A method uses `(String street, String city, String zip)` for addresses in multiple places. Refactor by introducing a small `Address` class and adjusting the method.
4. You have `customer.getCart().getAddress().getCity()`. Show how to fix it by introducing a method in one or more classes to reduce the chain. Provide code before and after.

Solutions

Part A

1. **Answer:** C. Refactoring improves internal structure without changing external behavior.
2. **Answer:** B. Extract smaller methods from a Long Method.
3. **Answer:** B. Replace conditionals with polymorphism.
4. **Answer:** A. “Feature Envy” is using another class’s data more than your own.
5. **Answer:** B. Testing incremental changes ensures you don’t break behavior.
6. **Answer:** C. God Classes do too many things (lack of cohesion).
7. **Answer:** B. Data Clumps are sets of parameters/fields always used together.
8. **Answer:** B. Introduce parameter objects or domain classes.
9. **Answer:** C. Polymorphism typically replaces or reduces type-based conditionals.
10. **Answer:** A. A message chain is e.g. `order.getCust().getX().getY()...`.

Part B.

1. Example smells: *Long Method*, *Primitive Obsession*, *Switch Statement.* Each reduces maintainability. *Long Method* is tough to read/test, *Primitive Obsession* lacks domain objects for clarity, *Switch Statement* breaks open/closed principle.
2. *Long Method* is an excessively big function. Typically simpler to fix by extracting smaller methods. *Large Class* lumps many responsibilities. Breaking it up can be more challenging since it might require multiple new classes or rethinking the design.
3. `(x, y, width, height)` repeated is a typical data clump in a graphics library. We fix it by creating a `Rectangle` or `Dimension` class bundling these fields. Helps avoid mistakes and clarifies usage.
4. Example code: `receipt.getOrder().getCustomer().getAddress().getCity()`. Replacing it with `receipt.getCityOfCustomer()` or a helper method in `Order` or `Customer` shortens the chain. Improves encapsulation.
5. Contradicts the idea that each class handles its own data/logic. If a `Billing` class constantly uses `Order` fields more than its own, it belongs in `Order`. This fosters a healthy distribution of responsibilities.
6. *Refactoring*: small steps preserving behavior. *Rewrite*: start from zero, discarding old code. If code is irredeemably messy or shifting to a new platform, rewriting may be best. Otherwise, refactoring is generally less risky.

Part C

1. **Long Method → Extract Method**
   - **Before**:

```
class InvoicePrinter {
    public int printInvoice(Order order) {
        // 1) Validate order
        if (!order.isValid()) { return -1; }
        // 2) Calculate tax
        double tax = order.getTotal() * 0.08;

        // 3) Save to DB
        Database.save(order);

        // 4) Logging
        Logger.log("Invoice for order " + order.getId() +
" printed.");

        // 5) Output final invoice
        System.out.println("Order ID: " + order.getId());
        System.out.println("Tax: " + tax);
        System.out.println("Final: " + (order.getTotal()
+ tax));
        return 0;
     }
     }
```

- **After**:

```
class InvoicePrinter {
   public int printInvoice(Order order) {
       if (!validateOrder(order)) return -1;
       double tax = calculateTax(order);
       persistOrder(order);
       logInvoice(order);
       outputInvoice(order, tax);
       return 0;
   }

   private boolean validateOrder(Order order) {
       return order.isValid();
   }
   private double calculateTax(Order order) {
       return order.getTotal() * 0.08;
   }
   private void persistOrder(Order order) {
       Database.save(order);
   }
   private void logInvoice(Order order) {
       Logger.log("Invoice for order " + order.getId() +
" printed.");
   }
   private void outputInvoice(Order order, double tax) {
       System.out.println("Order ID: " + order.getId());
       System.out.println("Tax: " + tax);
       System.out.println("Final: " + (order.getTotal()
+ tax));
   }
}
```

Explanation: We extracted smaller, well-named methods for each step.

2. **Switch Statement → Polymorphism**
   - **Before**:

```
    public double shippingCost(String shippingType,
double weight) {
        switch (shippingType) {
            case "AIR": return weight * 2.0;
            case "SEA": return weight * 1.0;
            case "LAND": return weight * 1.5;
            default: return 0.0;
        }
    }
```

   - **After**:

```
    interface Shipping {
        double calculateCost(double weight);
    }
    class AirShipping implements Shipping {
        public double calculateCost(double weight) {
return weight * 2.0; }
    }
    class SeaShipping implements Shipping {
        public double calculateCost(double weight) {
return weight * 1.0; }
    }
    class LandShipping implements Shipping {
        public double calculateCost(double weight) {
return weight * 1.5; }
```

```
   }

   // usage
   public double shippingCost(Shipping shippingStrategy,
double weight) {
       return shippingStrategy.calculateCost(weight);
   }
```

Explanation: We can add new shipping types without modifying the main method.

3. **Refactor "Primitive Obsession"**
   - **Before**:

```
   public void setAddress(String street, String city,
String zip) {
       // store them individually
   }
```

   - **After**:

```
   class Address {
       private String street;
       private String city;
       private String zip;
       // constructor, getters, etc.
   }

   public void setAddress(Address address) {
       // store the object
   }
```

Explanation: Replacing multiple string parameters with a domain class `Address`.

4. **Message Chain**
   - **Before**:

```
   String city =
order.getCustomer().getAddress().getCity();
```

   - **After**:

```
   // Introduce a method in Order
   class Order {
       Customer customer;
```

```
        public String getCustomerCity() {
            return customer.getCity(); // hide address
 detail
        }
    }
    // usage
    String city = order.getCustomerCity();
```

Explanation: We hide the chain behind a new method.