Refactoring and code smell

- Refactoring improves the internal design of the code while keeping its functionality the same
- Make code cleaner, maintainable, readable
- Remove duplication, simplify complex logic and improve structure.
- Verify no change in external behaviour by
  → testing, code analysis, carefulness.

□ Why do we refactor?

→ Deliver more business value faster
→ ease to maintainance and understanding
→ Improve software design.
→ Facilitate change and flexibility
→ Increase Reusability
→ Combat 'Bit Rot'
→ Minimize Technical debt.
→ Fix broken Windows
→ Help find bugs
→ Maintain development speed
→ Write for people, not for compiler.

- When should we Refactor?
  → To add new functionality
  → When existing code is not understandable
  → When we have to improve the design
  → To find bugs, remove duplicacy
  → During code reviews (improve readability)

- Differentiate between adding a function in a code and refactoring a function :-

Adding Function	Refactoring Function
To add new capabilities or features to the system	To improve internal structure of existing code without changing its behaviour.
Introduces new functionality to the program.	Only structure and readabili are improved.
New tests are written for new feature.	Existing test should still pass.
Goal is to get the test working	Goal is to restructure code to remove redundancy

- How do we refactor?
  → Look for Code smells (duplicated code, long methods, large ~~meth~~ classes, poor naming etc)
  → Apply refactoring techniques, when it stinks (renaming variables, simplyfying logic, etc).

Common Code smells:

① Inappropriate Naming:

→ Names given to variables and methods should be clear, descriptive and meaningful!

Bad Naming	Good Naming
private string s;	private string salary;
public double calc(double s)	public double calculateTax(double salary)
✗ hard to read and understand	✓ makes code maintable and
✗ Increase risk of misuse, bugs	self-explanatory

② Comments:

Try to refactor so the the code itself make the code self-documenting / intention-revealing because too many comments reduce the readability and maintainability.

Soln: ① Extract method: move block of code in a well-named method

② Rename method / variable that describe intent.

③ Introduce Assertion to state assumptions instead of comenting them.

Example:

Smelly code:

```
def process_items(items):
    // items should not be empty
    total = 0
    for item in items:
        total += item
```

Refactored code (with Assertion)

```
def process_items(items):
    assert len(items) > 0, "items should
    not be empty"
    total = 0
```

Solutions of Commenting :-

smelly → refactored.

✳ Rename Method:

Customer
getinvedlmt...

⇒

Customer
getInvoiceCreditLimit

✳ Extract Method: move parts of method into smaller, well-named methods.

→ Take piece of code inside a method and move it into a new method with meaningful name.

Smelly:

```
void PrintOwning(double amount){
    PrintBanner();
    //print details
    Sout("name:"+name);
    sout("amount:"+amount);
}
```

refactoring:

```
void PrintOwning
        (double amount) {
    PrintBanner();
    PrintDetails(amount);
}
void PrintDetails(double
                  amount){
    sout("name:"+name);
    sout("amount:"+amount);
}
```

✳ Introduce Assertion:

Smelly:

```
double getExpenseLimit(){
//should have either expense
limit or a primary project
return (_expenseLimit != NULL_EXPENSE)?_expenseLimit:
  _primaryProject.GetMemberExpenseLimit();
};
```

Refactoring:

```
double getExpenseLimit(){
  Assert(_expenseLimit != NULL_EXPENSE||_primaryProject!=null,
     "Both Expense Limit and Primary Project must
not be null");
```

```
return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :
    _primaryProject.GetMemberExpenseLimit( );
}
```

✓③ **Long Method:** Long method make it complex and hard to quickly understand.

Problem : X Hides behaviour
X Leads to duplicate code
X Hard to maintain, test, debug
X Violates good readability and simplicity

Soln: ✓ Extract Method (Pg-45-46)
✓ Replace Temp with Query
✓ Introduce Parameter object
✓ Preserve Whole Object
✓ Replace Method with Method Object
✓ Decompose conditional

⊛ **Replace Temp with Query:**

Smelly code

```
double basePrice = _quantity * _itemPrice;
                                // temp variable
if(basePrice > 1000){
    return basePrice * 0.95;
}else{
    return basePrice * 0.98;
}
```

Here, temp variable basePrice hides the meaning of calculation. It also repeat calculation risk errors

Refactor

```
if (getBasePrice() > 1000)
{ return getBasePrice() * .95
}else{
return getBasePrice() * .98
}
double getBasePrice() {
  return _quantity * _itemPric
} no temp needed. It make
```

(*) Introduce Parameter Object: All related parameter are grouped into a single object

Customer
amountInvoice(start:Date, end:Date) amountReceived(" " " ") amountOverdue(" " " ")

⇒

Customer
amountInvoice(DateRange) amountReceived(") amountOverdue(")

Example:

Smelly:

```
Public double calculatePrice
  (int quantity, double itemPrice,
   double taxRate, double discount){
  double basePrice = quantity * itemPrice;
double discount = basePrice *(1-discount);
return discount*(1+taxRate);
}
```

Refactor:

```
class PriceData {
  int quantity;
  double itemPrice, taxRate,
   discount;
PriceData(int quantity,
double itemPrice ....){
 this.quantity = quantity;
 ..... }
public double
 calculatePrice(PriceData
                 data){
double basePrice = data.quantity
    * data.itemPrice;
double discount = basePrice*
(1-data.discount);
return discount*(1+data.
                 taxRate);
}
```

When a method has many parameter, combine them into a single object to simplify method calls and make the code cleaner and safer.

(*) Preserve Whole Object:

(So the method can handle it in details)

Instead of passing parts of obj, pass entire object

Smelly:

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

Refactor:

```
withinPlan = plan.withinRange(daysTempRange());
```

(*) Replace Method with Method Object:

Smelly:

```
//class Order
double price(){
    double primaryBasePrice;
    double secondaryBasePrice;
    double teritaryBasePrice;
    ... //long computation
    return totalPrice;
}
```

The method is long and uses many local variable, so it is hard to maintain/extend.

Refactored

```
//class Order
double price(){
    return new PriceCalculator(this).
                                compute();
}//New Method object
class PriceCalculator {
    Order order;
    double primaryBasePrice, seco....;
PriceCalculator(Order order) {
    this.order = order; }
double compute() {
    ----//long computation
    return totalPrice; }
}
```

After refactoring, the long computation is now in its own class, all local variables become fields of the object. Original price() method is, short, clean, readable.

∴ long, complex method ⟶ Separate class

(*) Decompose Conditional:

Find long or complicated if-then-else that mix multiple condition and extract them.

Smelly:

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else
    charge = quantity * _summerRate;
```

Refactor:

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge(quantity);
```

✓ ④ **Long Parameter List:** Methods that take too many parameters produce code that is awkward and difficult to work with.

Soln: ✓ Preserve Whole Object
✓ Introduce Parameter Object
✓ Replace Parameter with Method

* **Replace Parameter with Method:**

Smelly:

```
public double getPrice(){
  int basePrice = quantity * itemPrice;
  int discount = getDiscout();
  double finalPrice = discountPrice(basePrice, discountPrice);
  return finalPrice;
}
private int getDiscount(){
  if (quantity > 100) return 2;
  else return 1;
}
private double discountPrice
      (int basePrice, int discount){
  if (getDiscount == 2)
      return basePrice * 0.1;
  else
      return basePrice * 0.05;
}
```

In the code discountPrice takes the discount as a parameter but it can get itself using getDiscount which is redundant

Refactor:

```
public double getPrice()
{
int basePrice = q * itemPrice;
double finalPrice;
finalPrice = discountPrice
        (basePrice);
return finalPrice;
}
private int getDiscount(){
 if(quantity > 100)
   return 2
 else return 1; }
private double
 discountPrice(int basePrice){
//get discountLevel directly
if (getDiscount() == 2)
   return basePrice * 0.1;
else return basePrice * 0.05;
}
```

✓ (5) **Feature Envy:** A method that seems more interested in some other classes than its own class.

- It happens when a method uses more data or methods from another class than its own class.

Problem:

→ frequently calls getters of another class
→ perform calculations mostly on another class data
→ It could logically belong to other classes.

Soln:

✓ Move Field → move the field where it is used mostly
✓ Move Method → move the method to the class where it envies
✓ Extract Method

Smell:

```
class Order {
    privat Customer customer;
public String getCustomerAddress(){
  return customer.getName()+"\n"+
         customer.getStreet()+"\n"+
         customer.getCity()+"\n"+
         customer.getZip();
} }
class Customer {
    private String name, street,
    city, zip ;
} // getters of all fields
```

Refactor:

```
class Customer {
 private String name,
   street , city, zip;
 public String
   getAdress() {
 return name+"\n"+
 street+"\n"+city+zip;
}}
class Order {
  private customer customer;
public String getCustomerAddres
{ return customer.getAddress();
}
```

⑥ **Dead code**: Code that is no longer used in a system or related system is Dead code.

Problem:	Soln:
✗ Increase complexity, readability	✓ Delete
✗ Accidental changes → bugs	
✗ More dead code	

⑦ **Duplicated Code:**

- It occurs when same/similar code exist in multiple places.

Problem: ① Obvious/blatant duplication: Copy and Paste

② Subtle/non-obvious duplication: Similar logic or algorithms accross classes with small variation.

Soln: ✓ Extract Method
✓ Pull Up Field
✓ Substitute Algorithm
✓ Form Template Method.

Levels of Duplication:

① Literal Duplication: Exact same code in multiple places.

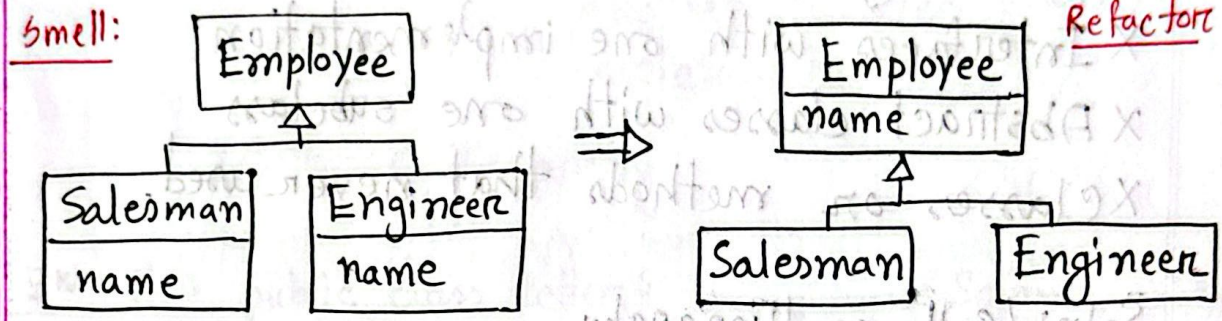
② Semantic Duplication: Similar code/logic, not exactly identical

③ Data Duplication: Same data or constants appear in multiple places.

④ Conceptual Duplication: Different algorithms achieve same goal (example - Bubblesort and QuickSort)

⑤ Logical Steps - Duplication: Same sequence of logical steps is repeated in multiple scenarios. Repeated steps of validation in various points.

✱ Pull up Field: move common field from subclass to superclass. It simplifies inheritence hierarchy.



✓ Removes duplication
✓ Make inheritence clear, simplify maintenance
✓ Increase cohesion of superclass

✱ Form Template Method: When we have two or more methods in different subclasses that perform same thing, we can create template method in saperclass to define the common structure.

✱ Substitute Algorithm: When method's logic is too complex, duplicated or in efficient, we can replace it with a simpler or clearer algorithm.

Example: Replace manual loop with buit-in method

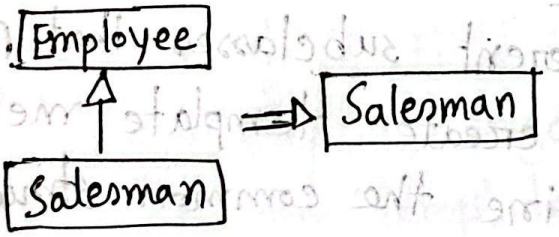
```
int findLargest(int[] numbers){
    return Arrays.stream(numbers).max().getAsInt();
}
```

⑥ **Speculative Generality:** It happens when developers add flexibility or generally "just in case" for future needs that don't exist yet.

Problem: X Parameters for future extension but never used.
X Over-engineed inheritence hierarchies
X Interfaces with one implementation
X Abstract classes with one subclass
X Classes or methods that never used.

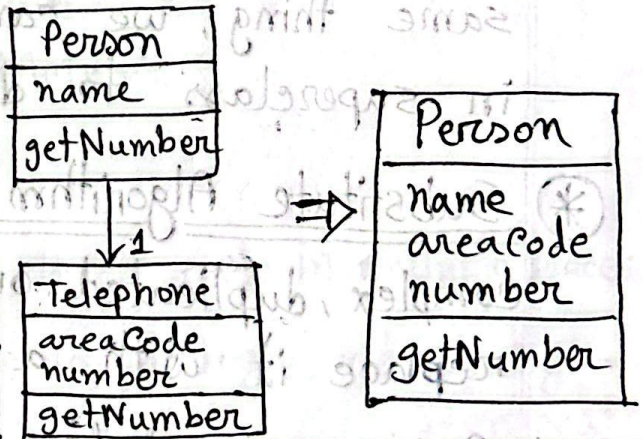
Soln: √Collapse Hierarchy
✓ Inline Class
✓ Remove Parameter.

⊛ **Collapse Hierarchy:**



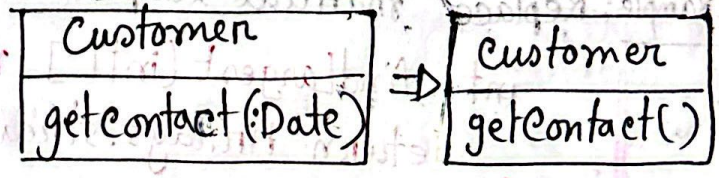
Remove unnecessary inheritence when superclass or subclass duplication adds no value.

⊛ **Inline Class:** Merge class that is not pulling its weight into another class



⊛ **Remove Parameter:**

Delete parameters that are not used.



✓(7) **Lazy class**: A class that does not do enough work to justify its existence.

In good OOD, each class should have clear purpose and responsibility.

Soln: ✓Inline Class → Move its functionality into another related class, then delete it.

✓Collapse Hierarchy

Example: public class Letter {

Smelly:
```
private final String content;
public Letter (String content){
    this.content = content;
}
public String getContent(){
    return content;
}
}
```

Soln: Here, Letter does nothing. just holds a string. If other classes only use it to access a string, it's better to inline it into those class.

✓(8) **Refused Bequest**: It occurs when a subclass inherits ↑methods or fields from a superclass but doesn't need them.

Problem:
- x Sub class overrides inherited methods with empty bodies
- x subclass throws exception, for inherited methods; it violates Liskov SP
- x Subclass doesn't use fields/methods it inherits
- x Hierarchy doesn't make sense.

Solution:
- ✓ Push down Method
- ✓ Push down Field

✓ (9) **Black Sheep:** a subclass / method that doesn't fit well with its family.

A subclass that is noticably different from other methods in the class.

```
public class StringUtil {
    ps String pascalCase(String string){
        return ...
    }
    public static ll camelCase(String string){
        return ...
    }
    public static String numberAndNoun(int Number, String noun)
{ return ... }
    public static string extractCommand(Map parameter){
        return ((String[] parameter.get("command"))[0];
    }
}
```

Blacksheep

Here, Every method is string formatting but extractCommand is not about String manupulation

✓ (10) **Primitive Obsession:** It occurs when primitive data types (like int, float, double, String etc) are used to represent higher level concept that should be modeled as object or classes.

Soln: ✓ Extract Class
✓ Replace Data with Object
✓ Replace code with Class
✓ Replace Array with Obj
✓ Introduce Parameter Object

Refactored:

Smelly:

```
public class CompositeShape
{
  IShape[] arr = new IShape[100];
  int count = 0;
  public void Add(Ishape shape){
    arr[count++] = shape;
  }
public void Remove(IShape Shape){
  for(int i=0; i<100; i++){
    if(shape==arr[i]){
      //code to remove
    }
}}}
```

Refactored:

```
public class CompositeShape
{
  List<IShape> shapeList
    = new List<IShape>();
  public void Add(IShape shape)
{
  ShapList.Add(shape);
}
public void Remove(IShape
                   shape)
{
  shapeList.Remove(shape);
}
}
```

It uses a fixed-size primitive array, manual index tracking (count), manual logic to remove. After fixing, no manual counting, looping. Uses List, which abstracts away low-level array operations.

(*) Replace Array with Object:

Smelly:

```
String[] row = new String[2];
row[0] = "Liverpool";
row[1] = "15";
```

↓

Refactor: `performance row = new Performance("Liverpool","15");`

Arrays used to hold mixed (Liverpool, 15 etc) or unrelated data. After fixing by creating a class (performance) with name field, methods, code become cleaner.

(11) **OddBall Solution:** • subtly duplicate code

- same problem is solved in different ways in different parts of the system.
- It leads to — Inconsistent code behaviour, harder maintenance, hidden duplication.

Refactore

```
String BaseAction::processView()
{
  return process("ViewAction");
}
String LoadUserProfileAction{
  return processView();
}
UploadAction::process(
{  return processView();
}
String ShowLoginAction(){
  return processView();
```

Smelly:

```
String LoadUserProfileAction::process()
{  //some code
   return process("ViewAction");
}
String UploadAction::process(){
   //code
   return process("ViewAction");
}
String ShowLoginAction::process(){
   //code
   Action* viewAction = actionProcessor().get("ViewAction");
   return viewAction->process();
}
```

different mechanism to call "ViewAction" method which is inconsistent

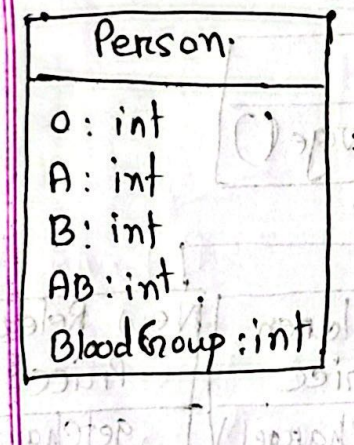
√Another Soln: Subtitute Algorithm.

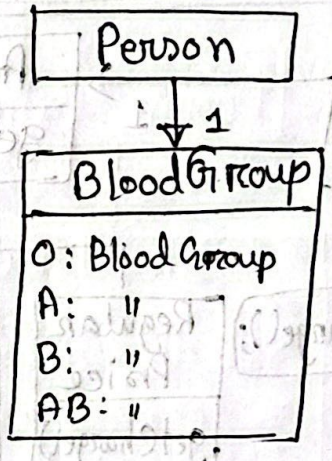
(12) **Large Class:** Class who suffer when they have too long responsibilities.

It also violates SRP, hard to maintain, understand.

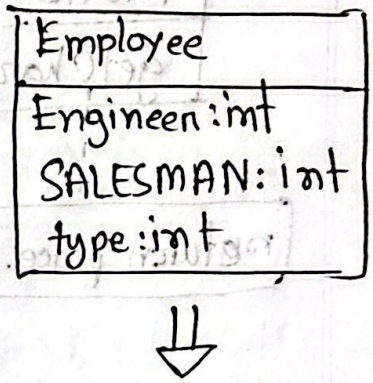
Soln:
- ✓ Extract Class
- ✓ Replace type code with class
- ✓ " " " " subclasses
- ✓ " " " " State/Strategy
- ✓ " Conditional " Polymorphism

(*) **Replace type code with class:**

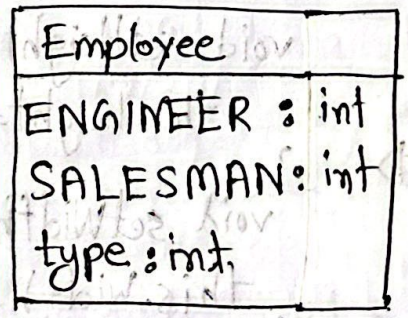


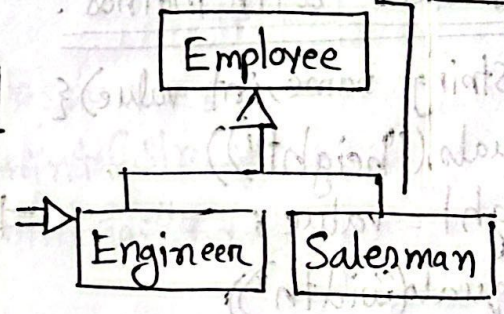


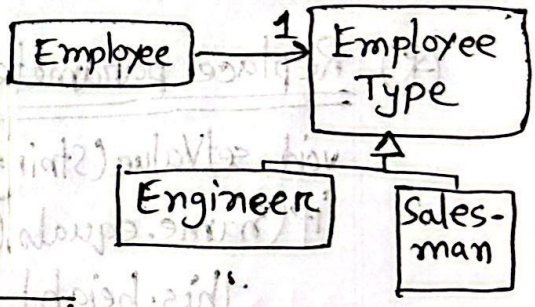
Replace Type with State/Strategy



(*) **Replace type code with subclasses:**



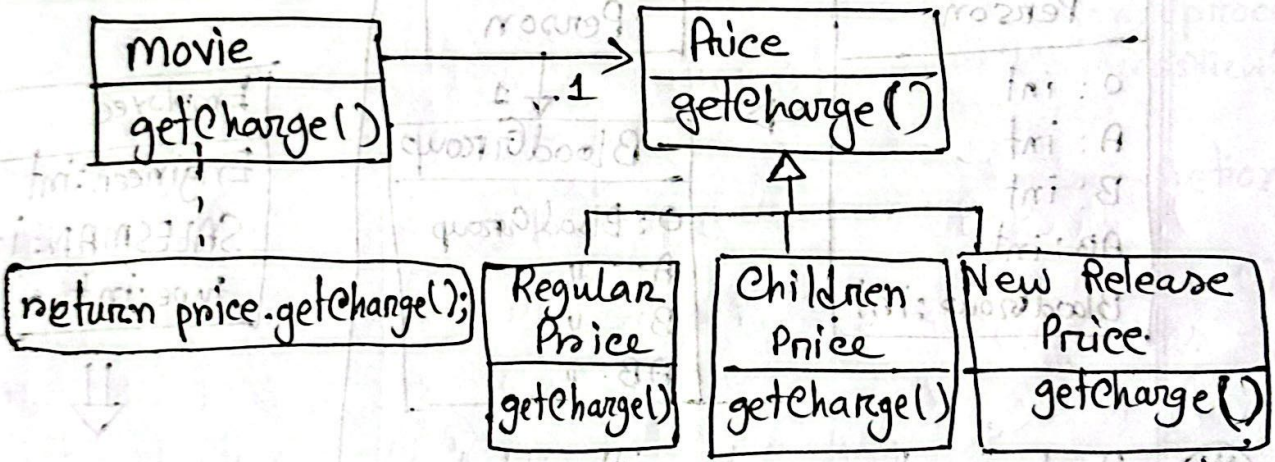




(13) **Switch Statement:** When code contains repeated switch statements (if..else if... else) duplicated across the system.

Soln: ✓Replace Type code With Polymorphism
✓ " " " " State/Strategy
✓ " Parameter " Explicit Methods
✓ Introduce null object

(*) **Replace Type with Polymorphism:**



(*) **Replace parameters with Method:**

```
void setValue(String name, int value){
if(name.equals("height"))
    this.height = value;
else if(name.equals("width"))
    this.width = value;
}
```

⇒

```
void setHeight(int h)
    this.height = h;
}
void setWidth(int w)
    this.width = w;
}
```

Singleton Pattern:

- Ensure that class has only one instance, while providing global access point to this instance

Steps:

1. Make default constructor private, to prevent other objects from using the new operator with the Singleton class.
2. Create a static creation method that acts as a constructor.

Example:

```
public class Printer {
    private static Printer instance = null;
    private Printer() {
        Sout("Printer created");
    }
    public static Printer getInstance() {
        if (instance == null) {
            instance = new Printer();
        }
        return instance;
    }
    public void print(String document) {
        sout("Printing:" + document);
    }
}

// Main class
Printer p1 = Printer.getInstance(); // create printer
Printer p2 = Printer.getInstance(); // Returns same printer
Sout(p1 == p2); // true - same instance
```

Design Pattern :-

Typical solutions of commonly occuring problems in software design

- √ Solves common, recurring software problems
- √ Improve code readability, maintainability, flexity.
- √ Reduces development time
- √ Promotes cleaner solution, reusability
- √ Improves collaboration, helps managing complex architecture