DESIGN PRINCIPLE-BASED REFACTORING: SCP

Speaking Code Principle (SCP)

The code should communicate its purpose.

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand"

-- Martin Fowler, 1999.

SCP-related Bad Smells

Comments: <u>Comments</u> in the code could indicate that the code communicates its purpose insufficiently.

```
// performs the transaction
  public void execute() {
     // get references to bank database and screen
     BankDatabase bankDatabase = getBankDatabase();
     Screen screen = getScreen();

     // get the total balance for the account involved
     double totalBalance = bankDatabase.getTotalBalance( getAccountNumber() );

     // display the balance information on the screen
     screen.displayMessageLine( "\nBalance Information:" );
     screen.displayMessage( "\n - Total balance:     " );
     screen.displayDollarAmount( totalBalance );
     screen.displayMessageLine( "" );
  }
```

Refactoring Techniques

Smell	Refactoring
Meaningless name	Use intention-revealing name
Complicated conditional expression	Introduce explaining variable
DD anomaly	Split temporary variable
Non const parameter	Remove assignments to parameters
literal	Replace magic number with symbolic constant
Implicit assertion	Introduce assertion
Unbalanced branch	Replace nested conditional with guard clauses
Error code	Replace error code with exception

Use Intention-revealing Name

You cannot understand what the function/variable does/stores.

```
float getValue(int m) {
  float r = 0.0F;
  if (m>= 3 && m<= 5)
     r = 0.2F;
  else if (m>= 6 && m<= 8)
     r = 0.5F;
  else if (m>= 9 && m<= 11)
     r = 0.2F;
  else
     r = 0.1F;
  return r ;
}
```

```
int main() {
  int x;
  cin >> x;
  float y = getValue(x);
  cout << "Discount Rate: "
       << y << endl;
}
```

Use Intention-revealing Name

Rename it that reveals its intention/purpose.

```
float getDiscountRate(int month) {
  float discountRate = 0.0F;
  if (month >= 3 && month <= 5)
    discountRate = 0.2F;
  else if (month >= 6 && month <= 8)
    discountRate = 0.5F;
  else if (month >= 9 && month <= 11)
    discountRate = 0.2F;
  else
    discountRate = 0.1F;
  return discountRate;
}
```

```
int main() {
  int month;
  cin >> month;
  float discountRate =
      getDiscountRate(month);
  cout << "Discount Rate: "
       << discountRate
       << endl;
}
```

Introduce Explaining Variable

You have a complicated expression in condition

```
float getDiscountRate(int month) {
  float discountRate = 0.0F;
  if (month >= 3 && month <= 5)
    discountRate = 0.2F;
  else if (month >= 6 && month <= 8)
    discountRate = 0.5F;
  else if (month >= 9 && month <= 11)
    discountRate = 0.2F;
  else
    discountRate = 0.1F;
  return discountRate;
}
```

Introduce Explaining Variable

Put the result of the expression in a temporary variable with a name that explains the purpose

```
float getDiscountRate(int month) {
  float discountRate = 0.0F;

  bool isSpring = month >= 3 && month <= 5 ;
  bool isSummer = month >= 6 && month <= 8 ;
  bool isFall = month >= 9 && month <= 11 ;

  if ( isSpring ) discountRate = 0.2F;
  else if ( isSummer ) discountRate = 0.5F;
  else if ( isFall ) discountRate = 0.2F;
  else discountRate = 0.1F;

  return discountRate;
}
```

Extract method(replace temp with query) can be considered

Split Temporary Variable

You have a temporary variable assigned to more than once, but not a loop variable nor a collecting temporary variable

```
public void printInfo(int width, int height)
{
  double temp = 2 * (width + height);
  System.out.println(temp);
  temp = width * height *;
  System.out.println(temp);
}
```

Split Temporary Variable

Make a separate temporary variable for each assignment

Any variable should be assigned with one responsibility
➔ cohesion

```
public void printInfo(int width, int height) {
  final double perimeter = 2 * (width + height);
  System.out.println(perimeter  );

  final double area = width * height *;
  System.out.println(area);
}
```

Remove Assignments to Parameters

The code assigns to a parameter

```
int getDiscount (int inputVal, int quantity, int yearToDate)
{
  if (inputVal > 50) inputVal -= 2;
  if (quantity > 100) inputVal -= 1;
  if (yearToDate > 10000) inputVal -= 4;
  return inputVal;
}
```

Remove Assignments to Parameters

Use a temporary variable instead

You can enforce this convention with the **final/const** keyword

```
int getDiscount (final int inputVal, final int quantity,
                 final int yearToDate) {
  int discount = inputVal;
  if (inputVal > 50) discount -= 2;
  if (quantity > 100) discount -= 1;
  if (yearToDate > 10000) discount -= 4;
  return discount;
}
```

Replace Magic Number with Symbolic Constant

You have a literal number with a particular meaning

Magic numbers are really nasty when you need to reference the same logical number in more than one place ➔ **shotgun surgery**

```
Public double getPotentialEnergy(
    final double mass, final double height) {

  return mass * 9.81 * height;
}
```

Replace Magic Number with Symbolic Constant

Replace the literal with a constant with meaningful name

```
private static final double GRAVITATIONAL_CONSTANT = 9.81;

public double getPotentialEnergy(
    final double mass, final double height) {

  return mass * GRAVITATIONAL_CONSTANT * height;
}
```

Introduce Assertion

A section of code assumes something about the state of the program

Sometimes the assumptions are stated with a comment

```
private static final double NULL_EXPENSE = -1.0;
private double expenseLimit = NULL_EXPENSE;
private Project primaryProject;

public double getExpenseLimit() {
  // should have either expense limit or a primary project
  return (expenseLimit != NULL_EXPENSE) ? expenseLimit:
    primaryProject.getMemberExpenseLimit();
}
```

Introduce Assertion

Make the assumption explicit with an assertion

```
private static final double NULL_EXPENSE = -1.0;
private double expenseLimit = NULL_EXPENSE;
private Project primaryProject;

public double getExpenåseLimit() {
  Assert.isTrue (expenseLimit != NULL_EXPENSE ||
    primaryProject != null);

  return (expenseLimit != NULL_EXPENSE) ? expenseLimit:
    primaryProject.getMemberExpenseLimit();
}
```

Replace Nested Conditional with Guard Clauses

A method has conditional behavior that does not make clear the normal path of execution

Some branch says, “This is rare, and if it happens, do something and get out”

```
public double getAdjustedCapital() {
  double result = 0.0;
  if (capital > 0.0) {
    if (intRate > 0.0 && duration > 0.0) {
      result = (income / duration) * ADJ_FACTOR;
    }
  }
  return result;
}
```

Replace Nested Conditional with Guard Clauses

Use guard clauses for all the special cases.

guard clause: If the condition is an unusual condition, check the condition and return if the condition is true

```
public double getAdjustedCapital() {
  if (capital <= 0.0)
     return 0.0;
  if (intRate <= 0.0 || duration <= 0.0)
     return 0.0;
  return (income / duration) * ADJ_FACTOR;
}
```

Single exit point

- Clarity is the key principle
- If the code is clearer with multiple exit points, use them.

Replace Error Code with Exception

A method returns a special code to indicate an error

Unix and C-based systems traditionally use a return code to signal success or failure of a routine

```
public int withdraw(final int amount) {
  // return -1 when withdrawal cannot be done
  if (amount > balance)
    return -1;
  else {
    balance -= amount;
    return 0;
  }
}
```

Replace Error Code with Exception

Throw an exception instead.

Exceptions clearly separate normal processing from error processing which makes programs easier to understand

```
public void withdraw(int amount) throws BalanceException {
  if (amount > balance) throw new BalanceException();
  balance -= amount;
}
```