Porządki w kodzie czyli nie tylko o refaktoringu cz. 3

Przyjrzyjmy się teraz metodzie boolMultipy oraz zmiennej lokalnej o nazwie sum. Zmienna ta jest deklarowana na samym początku metody, uzywana jest dużo później. Jest to nawyk, który pozostał jeszcze po językach proceduralnych (takich jak wczesne C przy PL/SQL), gdzie wymaga się zadeklarowania wszystkich zmiennych używanych w metodzie na samym początku. Na szczęście większość współczesnych języków programowania (w szczególności języków obiektów) nie ma tego ograniczenia. Zamiast tego podejścia sugeruję realizację zasady leniwej deklaracji zmiennych, którą można wyrazić za pomocą zdania

> *Deklaruj zmienne najpóźniej jak to tylko możliwe*

Warto to robić z bardzo prostego powodu – łatwiej będzie czytać kod, jeśli w zasięgu wzroku będziemy mieli operacje wykonywane na danej zmiennej. Przy bardziej złożonych metodach umieszczenie deklaracji na samym początku, może spowodować, że analizując dalszą część metody nie będziemy w stanie zorientować się czy zmienna była do tej pory przetwarzana czy nie i co wpłynęło na jej wartość.

Załóżmy, że przewinęliśmy ekran tak, że widzimy następujący kod:

```
        for( int i = 0; i < len; i++ ) {
            if ( this.get(i) && extendedBitSet.get(i) ) {
                sum++;
            }
        }
        return sum % 2 ;
```

Rodzi się we mnie natychmiast pytanie – a co to za suma? Czy działo się z nią coś wcześniej? Czy może jej wartość została pobrana z zewnątrz?

W przypadku jednej zmiennej jeszcze to nie jest duży problem, ale wyobraźmy sobie sytuację, kiedy takich zmiennych jest pięć. Śledzenie logiki takiej metody będzie bardzo trudne.

Bardzo prosta operacja przeniesienia deklaracji nieco niżej spowoduje, że kod będzie dużo bardziej czytelny:

```
    public int boolMultiply( ExtendedBitSet extendedBitSet ) {
        int len = 0;
        if( this.fixedLength < extendedBitSet.fixedLength ) {
            len = this.fixedLength;
        } else {
            len = extendedBitSet.fixedLength;
        }
```

```
        int sum = 0;
        for( int i = 0; i < len; i++ ) {
            if ( this.get( i ) && extendedBitSet.get( i ) ) {
                sum++;
            }
        }
        return sum % 2 ;
    }
```

Nieco bardziej złożona sytuacja jest w metodzie toByteArray. Jest ona niesamowicie trudna w analizie. Mi zajęło około 20 minut zrozumienie zasady jej działania. Teraz sobie wyobraźmy projekt złożony z tysiąca klas, z których każda zawiera tego typu metody. Zapanowanie nad takim kodem będzie graniczyło z cudem.
Spójrzmy na kod tej metody:

```
    public byte[] toByteArray() {
        int bytesNumber = 0;

        if ( fixedLength % 8 == 0 ) {
            bytesNumber = fixedLength / 8;
        } else {
            bytesNumber = fixedLength / 8 + 1;
        }
        byte [] arr = new byte[ bytesNumber ];
        for( int j = bytesNumber - 1, k = 0; j >= 0 ; j--, k++ ) {

            for( int i = j * 8 ; i < ( j + 1 ) * 8; i++ ) {
```

```
                if ( i == fixedLength ) {
                    break;
                }
                if ( get( i ) ) {
                    arr[ k ] += (byte) Math.pow( 2, i % 8 );
                }
            }
        }
        return arr;
    }
```

Metoda toByteArray zwraca bajtową reprezentacje ciągów bitowych – czyli ciąg bitowy o długości 14 bitów można zaprezentować za pomocą 2 bajtów, zaś ciąg bitowy o długości 23 bity można zaprezentować za pomocą 3 bajtów.

Algrotym można opisać w następujących krokach:

- określ liczbę bajtów,
- dla każdej ósemki bitów (lub kilku bitów w przypadku niepełnych bajtów) wykonaj następującą operację:
  - sprawdź wartość każdego bitu
  - jeśli bit ma wartość 1, dodaj odpowiednią potęgę dwójki (wynikającą z pozycji bitu w bajcie) do wyniku danego bajtu.

Dlaczegóżby nie wyrazić tego algorytmu w formie programistycznej? Często o tym się zapomina. Jako programiści

próbujemy w zwięzły sposób wyrazić nasze pomysły, co zazwyczaj prowadzi do bardzo nieczytelnych rozwiązań, których nie tylko inni ale i my sami nie jesteśmy w stanie zrozumieć w krótkim czasie. Ideałem jest dążenie do tego, aby jedno spojrzenie wystarczyło do odszyfrowania intencji twórcy. Nie potrzebujemy zagłębiać się w szczegóły realizacji algortmu, ważne byśmy wychwycili jego główną myśl. Spójrzmy na inną implementację metody toByteArray:

```
public byte[] toByteArray() {
    int bytesCount = computeBytesCount();
    byte [] byteArray = new byte[ bytesCount ];
    for ( int i = 0; i < bytesCount; ++i ) {
        int byteNumber = bytesCount - i - 1;
        byteArray[ byteNumber ] = computeByteValue( i );
    }
    return byteArray;
}
```

Najważniejsza zmiana, polega na bezpośrednim wyrażeniu algorytmu na ogólnym poziomie. Dzięki czemu jesteśmy w stanie w krótkim czasie zrozumieć intencję twórcy. Pomocnicza metoda computeBytesCount znajduje ilość bajtów nowej reprezentacji. W pętli dla każdego bajtu wykonywana jest operacja obliczania wartości bajtu i zapamiętywania wyniku. Czyż taki zapis nie jest dużo prostszy? Co z tego, że nie widać wszystkich elementów realizacji algorytmu. Bardziej dociekliwi zawsze mogą zajrzeć do metod computeBytesCount i computeByteValue.

Mogą one wyglądać następująco:

```
private byte computeByteValue( int byteNumber ) {

    int firstBitPosition = byteNumber * 8;

    int lastBitPosition = ( byteNumber + 1 ) * 8 - 1;


    byte byteValue = 0;

    for ( int i = this.nextSetBit( firstBitPosition );

        i >= firstBitPosition &&  i <= lastBitPosition;

                i = this.nextSetBit( i + 1 ) ) {

        int currentBitPosition = i - firstBitPosition;

        if ( get( i ) == true ) {

            byteValue += (byte) Math.pow( 2, currentBitPosition % 8 );

        }

    }

    return byteValue;

}


private int computeBytesCount() {

    int bytesCount = 0;

    if ( fixedLength % 8 == 0 ) {

        bytesCount = fixedLength / 8;

    } else {

        bytesCount = fixedLength / 8 + 1;

    }

    return bytesCount;

}
```

Warto zauważyć, że kod realizujący zadanie zbytnio się nie uprościł,

ale dużo łatwiej jest go teraz przeanalizować i zrozumieć. Teoria refaktoringu nazywa tego typu działania wyłuskiwaniem metody (ang. extract method). Prawdę mówiąc tutaj poszliśmy nieco dalej, gdyż zmodyfikowaliśmy nieco implementację algorytmu, tak aby stała się bardziej czytelna. Zwróćmy uwagę na wiersze:

```
int firstBitPosition = byteNumber * 8;

int lastBitPosition = ( byteNumber + 1 ) * 8 - 1;
```

Tak naprawdę są one wyodrębnieniem bardzo enigmatycznych wyrażeń z wiersza:

```
for( int i = j * 8 ; i < ( j + 1 ) * 8; i++ ) {
```

Czyż intencja w takim przypadku nie staje się oczywista? Moje wieloletnie doświadczenia doprowadziły mnie do wniosku:

Jawnie nazywaj złożone elementy kodu

zamiast

```
j * 8
```

napisz

```
int firstBitPosition = byteNumber * 8;
```

Zasadę te rozszerzam do instrukcji warunkowych. Często w kodzie możemy spotkać wyrażenia, za którymi kryje się pewna logika. Warto bezpośrednio wyrazić naszą intencję. Czytelność niesamowicie wzrasta.

Zamiast

```
if ( index >= 0 )
```

napisz

```
boolean isIndexInRange = ( index >= 0 );

if ( isIndexInRange )
```

Kod zaczyna się czytać jak książkę! Przecież programowanie jest dla ludzi. Ułatwiajmy zatem sobie życie.

A oto najważniejsza zasada, będąca kwintesencją powyższych rozważań:

> *Pisz kod w taki sposób, aby czytało się go jak powieść. Używaj jednoznacznych i jednocześnie prostych nazw. Realizowane operacje dziel na logiczne części i każdą implementuj w osobnych metodach.*

Myślę, że jak na jeden raz, wystarczy. Wystarczy, żeby zaostrzyć apetyty i wzbudzić ochotę na więcej. Żeby oprowadzić nieco po ogrodzie refaktoringu i jego przyległościach. Zapewniam, że to niesamowite miejsce i daje niesamowicie wiele radości i satysfakcji.

Poniżej zamieszczam ostateczną wersję kodu, który przechodzi załączone testy (oczywiście ze zmianą uwzględniającą niestatyczność metod merge i boolMultiply).

Końcowa postać przykładowej klasy (warto ją porównać z postacią początkową):

```
public class ExtendedBitSet extends BitSet {

    private int fixedLength = 0;

    public ExtendedBitSet( int size, String str ) {

        super( size );

        fixedLength = size;

        initializeBitSet( str );

    }
```

```
public ExtendedBitSet( String str ) {

    this( str.length(), str );

    initializeBitSet( str );

}


private void initializeBitSet( String str ) {

    int strLength = str.length();

    for( int i = 0; i < strLength; ++i ) {

        if ( str.charAt( strLength - 1 - i ) == '1' ) {

            set( i );

        }

    }

}


public void merge( ExtendedBitSet extendedBitSet ) {

    for ( int i = extendedBitSet.nextSetBit( 0 ); i >= 0;

                    i = extendedBitSet.nextSetBit( i + 1 ) ) {

        this.set( this.fixedLength + i );

    }

    this.fixedLength = this.fixedLength + extendedBitSet.fixedLength;

}


public int boolMultiply( ExtendedBitSet extendedBitSet ) {

    int len = 0;

    if( this.fixedLength < extendedBitSet.fixedLength ) {

        len = this.fixedLength;

    } else {

        len = extendedBitSet.fixedLength;
```

```
        }
        int sum = 0;
        for( int i = 0; i < len; i++ ) {
            if ( this.get( i ) && extendedBitSet.get( i ) ) {
                sum++;
            }
        }
        return sum % 2 ;
    }

    public byte[] toByteArray() {
        int bytesCount = computeBytesCount();
        byte [] byteArray = new byte[ bytesCount ];

        for ( int i = 0; i < bytesCount; ++i ) {
            int byteNumber = bytesCount - i - 1;
            byteArray[ byteNumber ] = computeByteValue( i );
        }
        return byteArray;
    }

    private byte computeByteValue( int byteNumber ) {
        int firstBitPosition = byteNumber * 8;
        int lastBitPosition = ( byteNumber + 1 ) * 8 - 1;

        byte byteValue = 0;
        for ( int i = this.nextSetBit( firstBitPosition );
            i >= firstBitPosition &&  i <= lastBitPosition;
```

```
                i = this.nextSetBit( i + 1 ) ) {
            int currentBitPosition = i - firstBitPosition;
            if ( get( i ) == true ) {
                byteValue += (byte) Math.pow( 2, currentBitPosition % 8 );
            }
        }
        return byteValue;
    }



    private int computeBytesCount() {
        int bytesCount = 0;
        if ( fixedLength % 8 == 0 ) {
            bytesCount = fixedLength / 8;
        } else {
            bytesCount = fixedLength / 8 + 1;
        }
        return bytesCount;
    }


    public String convertToBitString( int size ) {
        char [] resultArray = new char[ size ];
        for ( int i = 0; i < size; ++i ) {
            resultArray[ i ] = '0';
        }


        for ( int i = this.nextSetBit( 0 ); i >= 0; i = this.nextSetBit( i + 1 ) ) {
            resultArray[ size - 1 - i ] = '1';
```

```
        }
        return new String( resultArray );
    }

    public String convertToBitString() {
        return convertToBitString( this.fixedLength );
    }
}
```

Dodatek. Test przykładowej klasy.

```
public class ExtendedBitSetTest extends TestCase {
    public void testConstructorSizeAndString() throws Exception {
        assertEquals( "{0, 2}", new ExtendedBitSet( 10, "101" ).toString() );
        assertEquals( "0000000101", new ExtendedBitSet( 10, "101" ).convertToBitString() );
        assertEquals( "0000001011", new ExtendedBitSet( 10, "1011" ).convertToBitString() );
        assertEquals( "0010001011", new ExtendedBitSet( 10, "10001011" ).convertToBitString() );
        assertEquals( "{0, 1, 3, 7}", new ExtendedBitSet( 10, "10001011" ).toString() );
        assertEquals( "{0}", new ExtendedBitSet( 10, "001" ).toString() );
        assertEquals( "0000000001", new ExtendedBitSet( 10, "001" ).convertToBitString() );
        assertEquals( "00000000001", new ExtendedBitSet( 10, "001" ).convertToBitString( 11 ) );
    }
    public void testMerge() throws Exception {
        ExtendedBitSet extendedBitSet1 = new ExtendedBitSet( 10, "1" );
        ExtendedBitSet extendedBitSet2 = new ExtendedBitSet( 10, "1" );
        assertEquals( "0000000001", extendedBitSet1.convertToBitString() );
        assertEquals( "0000000001", extendedBitSet2.convertToBitString() );
```

```
        ExtendedBitSet mergedBitSet = ExtendedBitSet.merge( extendedBitSet1, extendedBitSet2 );
        String mergedString = mergedBitSet.convertToBitString();

        assertEquals( "00000000010000000001", mergedString );
        assertEquals( "{0, 10}", mergedBitSet.toString() );
        assertTrue( mergedBitSet.get( 0 ) == true );
    }

    public void testBoolMultiple() throws Exception {
        ExtendedBitSet extendedBitSet1 = new ExtendedBitSet( 3, "1" );
        ExtendedBitSet extendedBitSet2 = new ExtendedBitSet( 10, "1" );
        assertEquals( 1, ExtendedBitSet.boolMultiply( extendedBitSet1, extendedBitSet2 ) );

        extendedBitSet1 = new ExtendedBitSet( 1000, "1" );
        extendedBitSet2 = new ExtendedBitSet( 2, "1" );

        assertEquals( 1, ExtendedBitSet.boolMultiply( extendedBitSet1, extendedBitSet2 ) );

        extendedBitSet1 = new ExtendedBitSet( 10, "1" );
        extendedBitSet2 = new ExtendedBitSet( 10, "1" );

        assertEquals( 1, ExtendedBitSet.boolMultiply( extendedBitSet1, extendedBitSet2 ) );

        extendedBitSet1 = new ExtendedBitSet( 10, "1" );
        extendedBitSet2 = new ExtendedBitSet( 10, "10" );

        assertEquals( 0, ExtendedBitSet.boolMultiply( extendedBitSet1, extendedBitSet2 ) );
```

```
        extendedBitSet1 = new ExtendedBitSet( 10, "10" );
        extendedBitSet2 = new ExtendedBitSet( 10, "10" );

        assertEquals( 1, ExtendedBitSet.boolMultiply( extendedBitSet1, extendedBitSet2 ) );

        extendedBitSet1 = new ExtendedBitSet( 10, "110" );
        extendedBitSet2 = new ExtendedBitSet( 10, "110" );

        assertEquals( 0, ExtendedBitSet.boolMultiply( extendedBitSet1, extendedBitSet2 ) );
    }

    public void testToByteArray() throws Exception {

        ExtendedBitSet extendedBitSet = new ExtendedBitSet( "100000110" );
        byte[] toByteArray = extendedBitSet.toByteArray();

        assertEquals( 1, toByteArray[ 0 ] );
        assertEquals( 6, toByteArray[ 1 ] );

        extendedBitSet = new ExtendedBitSet( "10111111111" );

        toByteArray = extendedBitSet.toByteArray();

        assertEquals( 5, toByteArray[ 0 ] );
        assertEquals( -1, toByteArray[ 1 ] );
    }
}
```

O mnie

Mariusz Sieraczkiewicz

Trener, konsultant, menedżer projektów IT, coach. Założyciel zespołu programistów Equilibrium. Współinicjator JUGa Łódź. Autor artykułów o inżynierii oprogramowania. Współwłaściciel firmy szkoleniowej BNS IT. Szczęśliwy mąż :-)
http://www.linkedin.com/pub/2/a24/812

O BNS IT

BNS IT jest firmą szkoleniowo-doradczą zajmującą się **rozwijaniem i doskonaleniem zespołów programistycznych**.

Nasze usługi skierowane są do firm zatrudniających programistów. Pracujemy dla trzech grup klientów:

- firm zajmujących się wytwarzaniem oprogramowania
- firm, które posiadają zespoły programistyczne, lecz działają na innych rynkach niż IT
- firm integrujących systemy informatyczne

http://www.bnsit.pl/