Porządki w kodzie czyli nie tylko o refaktoringu cz. 1

Początkowo moim zamysłem było stworzenie artykułu o refaktoringu. Jednak im bardziej zastanawiałem się nad tematem, tym bardziej utwierdzałem się w przekonaniu, iż nie będę pisał tylko i wyłącznie o refaktoringu. Chodzi o coś znacznie istotniejszego, o przelanie bardzo rozległej wiedzy, a w zasadzie doświadczenia związanego z tworzeniem kodu. Kodu, który nie tylko działa, nie tylko jest dobrze zaprojektowany, ale przede wszystkim doskonale się czyta. Kiedy osiągamy tę umiejętność, stajemu u progu profesjonalizmu. Programistycznego profesjonalizmu.

Zatem będzie to artykuł między innymi o refaktoringu, ale wzbogacony o zbiór przemyśleń, sugestii, czasami również wątpliwości, którą mają pobudzić Cię, Czytelniku, do refleksji, zweryfikowania swoich programistycznych poczynań. Wierzę, że spowodują cały proces zmian – wprowadzenia nowych, dobrych nawyków.

Przede wszystkim czytelność

Programowanie bardzo szybko ewoluuje. Pamiętam jeszcze dość dobrze czasy, kiedy rozpocząłem swoją przygodę z kodowaniem jakieś dziesięć lat temu. Programy pisało się wtedy całkiem inaczej. Ceniono pomysłowość, zwięzłość i enigmatyczność. Im kod był bardziej niezrozumiały, tym programista był lepszy.

Jednak z czasem systemy informatyczne stawały się coraz bardziej skomplikowane, wymagały coraz większej wiedzy i co najważniejsze, stały się produktem pracy zespołowej. Obecnie pojedynczy programista nie jest w stanie zdziałać zbyt wiele. Być

może stworzy rozbudowany program desktopowy, natomiast nie będzie w stanie w wystarczająco skończonym czasie stworzyć rozproszonego systemu, opartego o architekturę trójwarstwową, zapewniającego odpowiedni poziom bezpieczeństwa, umożliwiającego zarządzanie prawami dostępu do wybranych części aplikacji, realizującym wielojęzyczność itp. itd. Takie systemy tworzy obecnie kilkunastu lub kilkudziesięciu programistów, w zależności od wielkości projektu, przez kilka lub kilkanaście miesięcy. Programista przestał być nierozumianym przez nikogo indywidualistą, a stał się graczem zespołowym, nastawionym na współpracę.

Co za tym idzie, sposób kodowania też musiał się zmienić. Wyłonił się podstawowy postulat dotyczący kodowania:

Przede wszystkim czytelność

Istnieją przynajmniej trzy podstawowe powody, które potwierdzają ważność tego stwierdzenia:

- wymagania się zmieniają,
- programowanie to umiejętność zespołowa,
- projekty są zbyt duże, aby pojedyncza osoba była w stanie ogarnąć całość.

Z tych właśnie powodu w ciągu ostatnich kilku lat bardzo mocno rozwijają się takie techniki jak refaktoring, pisanie testów oraz zwraca się ogromną uwagę na standard kodowania.

To właśnie Czytelność będzie głównym bohaterem tego artykułu. Będzie on zawierać sugestie i przemyślenia, które ułatwią realizację powyższego postulatu. Niektóre wskazówki będą stanowić moją subiektywną opinię, inne będą wyrażać mądrość doświadczeń społeczności programistycznej. Oczywiście należy pamiętać o pewnej zasadzie: “Jedyną niezmienną zasadą jest to, że nie ma niezmienych zasad”. Uogólniając, należy stwierdzić, iż przedstawiane wnioski sprawdziły się w wielu sytuacjach, co nie znaczy, że są zasadne w 100% przypadkach. Dlatego należy uważnie się przyglądać pojawiającym się na co dzień problemom i odważnie stosować przytoczone wskazówki. Warto krytycznie spojrzeć na swoje nawyki lub ich brak i rozpocząć zmiany. Zatem do dzieła!

Poprzez przykład do celu

Analiza kodu mniej doświadczonych programistów, często doprowadzała mnie do zaskakujących spostrzeżeń, umożliwiających znalezienie źródła problemów młodych (ale również i tych doświadczonych) adeptów sztuki programowania. Dlatego artykuł ten oparty będzie o przykład nie najlepiej napisanej klasy, która będzie analizowana i stopniowo udoskonalana.

Celem, postawionym przed autorami poniższego kodu, było zaimplementowanie klasy pochodnej klasy java.util.BitSet (wektora bitowego) wzbogaconej o:

- możliwość konkatenacji,
- właściwość narzuconej długości wektora (pole length),
- specyficznego mnożenia dwóch wektorów bitowych polegającego na zwróceniu wartości 0, jeśli jedynki w obu wektorach bitowych powtarzają się na parzystej ilości miejsc,

oraz wartości 1, jeśli jedynki pokrywają się na nieparzystej ilości miejsc,

- operację zamiany wektora na ciąg znakowy (w określonym z góry formacie),
- operację zamiany wektora w ciąg bajtów.

Pragnę zaznaczyć, iż treść przykładu nie ma tu większego znaczenia. Przytoczony kod służy tylko jako ilustracja często występujących niedoskonałości programistycznych. Ponadto, ponieważ nieodłączną częścią refaktoringu są testy, sprawdzające testowany kod, jako dodatek do artykułu została zamieszczona klasa testowa do analizowanej klasy.

Oto zaproponowana implementacja nowej wersji wektora bitowego:

```
import java.util.* ;

public class ExtendedBitSet extends BitSet {

    int length ;

    public ExtendedBitSet(int size, String str) {

        super(size) ;

        length = size ;

        int strLength = str.length();

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

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

        }

    }

    public ExtendedBitSet(String str) {

        this( str.length(), str );
```

```
        int strLength = str.length();
        for(int i = 0; i < strLength; ++i) {
            if(str.charAt( strLength - 1 - i) == '1') set(i) ;
        }
    }
    public static ExtendedBitSet merge(ExtendedBitSet a, ExtendedBitSet b) {
        StringBuffer str = new StringBuffer(a.convertToBitString() + b.convertToBitString()) ;
        return new ExtendedBitSet(a.length + b.length, str.toString()) ;
    }
    public static int boolMultiply(ExtendedBitSet a, ExtendedBitSet b) {
        int sum = 0 ;
        int len ;
        if(a.length < b.length) len = a.length ;
        else len = b.length ;
        for(int i = 0; i < len; i++) {
            if (a.get(i) && b.get(i)) sum++ ;
        }
        return sum % 2 ;
    }
    public byte[] toByteArray() {
        int bytesNumber ;
        if(length % 8 == 0) bytesNumber = length / 8 ;
        else bytesNumber = length / 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 == length) break ;
```

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

W pierwszej kolejności spójrzmy na klasę całościowo. Jedna z pierwszych rzeczy, która rzuca się w oczy to fakt, że metody konkatenacji i mnożenia wektorów są statyczne. Jest to sprzeczne z bardzo ważną zasadą:

Twórz spójne interfejsy i klasy

Jeśli przyjrzymy się klasie bazowej BitSet, łatwo zauważymy, iż żadna publiczna metoda nie jest statyczna. Dostępne są m. in. niestatyczne metody or(Bitset), xor(Bitset), których celem jest modyfikacja obiektu na rzecz którego są one wywolywane (operacja na this), a nie udostępnienie metody zewnętrznej (statycznej), która tworzy nowy obiekt, będący efektem implementowanej operacji. Zatem obydwie metody (merge i boolMultiply) swoją postacią wprowadzają rozdźwięk w strukturze nowej klasy, prowadząc do niespójnego interfejsu klasy ExtendedBitSet. W tym przypadku utrzymanie spójności poprzez zamianę metod statycznych na metody niestatyczne, uprości używanie klasy ExtendedBitSet, gdyż będzie się z niej korzystać tak samo jak z klasy BitSet.

Istnieje jeszcze jedna zasada, którą warto przytoczyć analizując metody merge i boolMultiply:

Unikaj statycznych elementów w programowaniu

Elementy statyczne to pozostałość po programowaniu proceduralnym, gdyż statyczność oznacza globalność. A przecież jedną z konsekwencji programowania obiektowego jest zamykanie implementowanych funkcjonalności w autonomicznych i możliwie jak najbardziej niezależnych obiektach. Dlatego elementów statyczne używaj tylko wtedy, kiedy nie ma innego wyjścia lub kiedy informacja lub operacja ma rzeczywiście charakter globalny. Zatem używaj pól statycznych jako stałych, szczególnie stałych globalnych, zaś metod statycznych używaj dla operacji globalnych. Przykładem użycia metod i pól statycznych jest wzorzec Singletonu, jednak „wzorcowość” tego wzorca bywa kwestionowana

(http://c2.com/cgi/wiki?SingletonsAreEvil). Ponadto należy pamiętać, że metody statyczne nie są polimorficzne, co oznacza, że nie możemy dostarczyć ich alternatywnych implementacji oraz że nie możemy ich zastępować za pomocą mocków. Zatem ich użycie powoduje usztywnienie kodu oraz utrudnia testowanie.

Zmieńmy zatem nieco przytoczony kod, zgodnie z pierwszymi dwoma regułami:

```
public void merge( ExtendedBitSet extendedBitSet ) {
    for ( int i = extendedBitSet.nextSetBit(0); i >= 0;
                        i = extendedBitSet.nextSetBit(i + 1) ) {
        this.set( this.length + i );
    }
    this.length = this.length + extendedBitSet.length;
}
public int boolMultiply( ExtendedBitSet extendedBitSet ) {
    int sum = 0 ;
    int len ;
    if(this.length < extendedBitSet.length) len = this.length ;
    else len = extendedBitSet.length ;
    for(int i = 0; i < len; i++) {
        if (this.get(i) && extendedBitSet.get(i)) sum++ ;
    }
    return sum % 2 ;
}
```

c. d. n.

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/