

# Object oriented analysis and design of IO bit and byte classes

Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Bedrijfsmanagement  
Software Engineering

September 24, 2009



## Methods convention in this course

In this course we will use methods that return primitive types and accept primitive types, matching as much as possible the types of the underlying layers.

Examples:

- byte 8 bit value
- short 16 bit value
- int 32 bit signed value
- long 64 bit signed value



## Operations per bit

- Bit has the following operations.
  - `set()`
  - `clr()` and
  - `isSet()`

First we will do the output part, which is simplest.





### Bit operations in words, set

To **set** a bit you use the **or** operator and 'one' at the bit position you want to set. 5 in the example. To do this you first calculate the *bit mask* from the bit number by shifting a 1 to the left by the number of bits. You can use the shift left operator (<<) for that as in  $1 \ll 5$ .

```
a =xxxxxxxx
m =00100000v
y =xx1xxxxx
```



### Bit operations in words, clear

To **clear** a bit you use the **and** operator and 'zero' at the bit position you want to clear. (Bit 4 in this example)

```
a =xxxxxxxx
m =11101111^
y =xxx0xxxx
```

Note that the m value in the clear operation is the bitwise inverse of the m value in the set operation.



### Bit testing

To test if a bit is set or clear (1 or 0) you can also use the mask value and the AND operation to isolate the bit of interest (bit 3 in this example).

```
a =xxxxxxxx
m =00001000^
y =0000x000
```



## Bit (un)equality

If you want to monitor (several) individual input bits and test them for changes, the **XOR** or *unequal* operation comes in handy. You can use the XOR operation to test which bits changed when comparing a previous value with the now actual value:

$$\begin{aligned}
 p &= 00101101 \\
 n &= \underline{01101100} \oplus \\
 c &= 01000001
 \end{aligned}$$

*p* is the previous or old value, *n* = the new value, then *c* (changed) has a one for each bit that differs between *p* and *n*. Note that the C (and Java) notation for the xor is ^.



## Hardware control from a virtual machine

Java is touted for its **compile one, run everywhere** feature. However, it cannot hold this promise once data exchange with the underlying hardware or even OS is involved. In that case you will have to speak the (native) tongue of the underlying services. In such cases the Java Native Interface approach must be used. This involves an interface (in Patterns you would call it an Adapter<sup>1</sup>). See also [http://en.wikipedia.org/wiki/Java\\_Native\\_Interface](http://en.wikipedia.org/wiki/Java_Native_Interface) and the links found there. JNI involves calling a C or C++ function from Java. Using this technology it is possible to create methods that read or write bytes or words on the hardware level.



<sup>1</sup>Even a very peculiar one: a Language Adapter

## Native methods

Native methods are methods adorned with the **native**. If the compiler finds this keyword, it will try to find a matching function in non JVM libraries, typically a DLL in Windows or a shared object (.so) under Unix.

The JDK brings along a few utilities in helping to write the stub code to interface to these C and C++ libraries.

One such tool is **javah**, which generates the stub code from your Java classes which use the native keyword.

Quit often the native methods are also defined as static.



## Example IO with JNI, Abstract operations:

The operations you need for io are quite simple and can be defined in a simple interface:

```
public interface IOOps {

    /**
     * Read a new value from this byte
     * @return the last value read
     */
    int read();

    /**
     * Write a value
     * @param newValue the value to write
     */
    void write(int newValue);

}
```



## Example io with JNI, native layer interface

To implement this for e.g. a parallel addressable IO port you can use this:

```
/**
 * Create a IOImpl associated with hardware address and
 * initial value 0.
 * @param a address of the port.
 */
public IOImpl(int a) {
    this(a, 0);
}

public int read() {
    return hhread(address);
}

public void write(int newValue) {
    hwwrite(address, shadow = newValue);
}
```



## JNI interfacing

The JNI interface is created by using the native annotation and loading the library that implements the method natively (in C or C++).

```
/**
 * Read a byte from a hardware address
 * @param the (hardware) address of the hw byte
 * @return the value read
 */
private static native byte hhread(int address);

/**
 * Write a byte to the hardware address
 * @param address to be written
 * @param value to be written
 */
private static native void hwwrite(int address, int value);

/*
 * Do the magic required to load my hwio C-library
 */
static {
    System.loadLibrary("hwio");
}
```



## The read-back problem

**IO is not memory:** In memory we expect to be able to read back what we wrote at a location. Otherwise it would be *memory!*?. In IO this is not the normal case. Writing to input is a silly operation. Compare it to read only memory. If you would want read back from output, then the hardware designer has to make a choice: read back what is actually seen at the output pins or read back what you wrote.

In the **first** case you would read back the value of what is written combined with the effects of the external load. You might end up writing a one and still reading 0. In the **second** case you need to add a *register* which holds the value you wrote last. This adds costs to the hardware for something the software could do almost for free.

Many pieces of IO hardware have the input and output port located at the same address, so reading is always the input operation and writing is always the output operation. In most cases **in** and **out** have no true relation. (E.g. Input being buttons, output lamps and motors).

**In most output hardware, you cannot read back what you wrote.**



## Let there be shadow

The solution to the read back problem is to maintain an up to date copy of the last value written to the output. The traditional name for such a value is **Shadow**.

**101100101**

By maintaining a shadow value per output word you can update individual bits without affecting the surrounding bits. Each bit can have its individual state and responsibility.

Note, that the shadow value should be maintained per IO word, not per bit. The bits in a word share the same shadow value. This implies we should store the shadow value in the class that does the *word*-write operations, the **IOWriter** class.



## Bit writes using shadow

When designing a Output Bit implementation with the as defined on sheet 3 we have to make several choices:

- ① Where to decide which logical operation to choose to set (OR) or clear (AND not) the bit
- ② Where and when to compute the mask value
- ③ How many methods to provide in the IOWriter interface

The answer to question 1 is not quite clear, but from the software engineering standpoint the answers to questions 2 and 3 are simple: **Do the calculations in as few places (DRY principle) and as few times (performance) we can.**



## Implementation 1

All computations in the writer class;

### Bit Ops implementation

```
private final IOWriter iowriter;
private final int bitNr;

public BitOpsImpl1(IOWriter iow, int bitnr){
    this.iowriter=iow;
    this.bitnr = bitnr;
}

public void set() {
    iowriter.setBit( bitNr );
}

public void clear(){
    iowriter.clearBit( bitNr );
}

public void set(boolean v) {
    if (v) set();
    else clear();
}

public boolean isSet() {
    return iowriter.testBit( bitNr );
}
```

### Required IOWriter Support

```
public void set(int bitNr){
    write(shadow != (1<< bitNr));
}

public void clear(int bitNr){
    write(shadow &= ~(1<< bitNr));
}

public boolean testBit(int bitNr) {
    return 0 != (shadow & (1<< bitNr));
}
```



## Implementation 2

Mask calculation and operation decision in Bit.

### Bit Ops implementation

```
private final IOWriter iowriter;
private final int bitNr;
private final int mask;

public BitOpsImpl2(IOWriter iow, int bitnr){
    this.iowriter=iow;
    this.bitnr = bitnr;
    this.mask = 1<< this.bitnr;
}

public void set(){
    iowriter.or(mask);
}

public void clear(){
    iowriter.and(~mask);
}

public void isSet(){
    return 0!=(iowriter.read() & mask);
}
```

### Required IOWriter Support

```
// for BitOpsImpl2
public void or(int v) {
    write( shadow |= v );
}

public void and(int v) {
    write( shadow &= v );
}

public boolean read() {
    return shadow & mask;
}
```



## Implementation 3

Mask and value decisions in Bit.

### Bit Ops implementation

```
private final IOWriter iowriter;
private final int bitNr;
private final int mask;

public BitOpsImpl2(IOWriter iow, int bitnr){
    this.iowriter=iow;
    this.bitnr = bitnr;
    this.mask = 1<< this.bitnr;
}

public void set(){
    iowriter.or(mask);
}

public void clear(){
    iowriter.and(~mask);
}

public void isSet(){
    return 0!=(iowriter.read() & mask);
}
```

### Required IOWriter Support

```
public void writeMasked( int v, int mask ){
    int ms = shadow & ~mask;
    int mv = value & mask;
    write(shadow = ms | mv);
}
```



### Implementation 3 wins

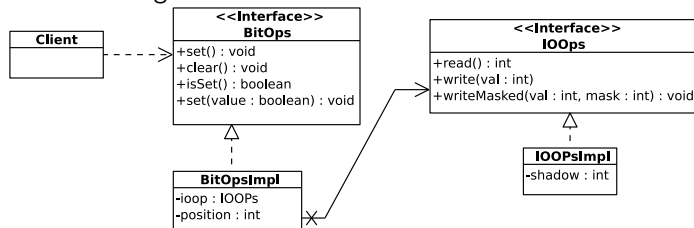
Because:

- The `BitOps.set` and `BitOps.clear` operations are dryly implemented using `BitOps.set(boolean v)`.
- The `IOWriter` implementation is a tad more complex in the `writeMasked` operation, but is otherwise optimally DRY, as all operations are done in one place and all decisions (if/else) are thrown out. Even the `write` method in `IOWriter` could be defined using the `writeMasked` operation. How? Quiz. (Think of how `write` is implemented).



### Class diagram for bitwise output

The class diagram so far:



The responsibilities of the classes are `BitOps` defines the bit operation contract, `BitOpsImpl` implements the operations using the methods defined in the `IOOps` type (contract) which are implemented by the `IOOpsImpl` class. The `IOOpsImpl` class is the class that implements the lowest, hardware specific layer. It is the only class that needs a specific implementation for the hardware/OS.

