

**Reducing complexity in Boolean expressions**  
**Aaldert Hofman**  
**Technology Consulting**  
**Cap Gemini Ernst & Young Nederland B.V.**

[Aaldert.Hofman@cgey.nl](mailto:Aaldert.Hofman@cgey.nl)

## **1 INTRODUCTION**

### **1.1 Intent**

The intent of this pattern is to enable business analysts, designers and programmers to reduce complexity in Boolean expressions in design and program, thus improving quality of programs.

### **1.2 Motivation**

A major cause of errors in programs is the fact that a human being isn't a computer: we do not think binary. Especially in program logic this causes problems. Every attempt to avoid these problems will help to improve the quality of programs.

The problem exists in many domains, ranging from avionics to calculating old age pensions. Complicated Boolean expressions cause problems in maintenance, testing and debugging, to name some of the more common areas.

### **1.3 Acknowledgements**

I want to thank my colleagues Ben Elsinga and Henk van Zuilekom who reviewed the paper. I owe special thanks to Henk, who provided me with the Excel spreadsheet accompanying this paper.

I would like to thank my EuroPLoP2003 shepherd Linda Rising who guided me through several iterations of this submission and provided many constructive comments. We argued whether this pattern is a pattern indeed. Well, find out yourself. Whether a pattern or not, we agreed that it's useful. So, it's worth writing up, worth shepherding and worth debating at a writers' workshop.

## 2 REDUCING COMPLEXITY IN BOOLEAN EXPRESSIONS

### 2.1 Context

One of the major structuring elements in software programs is “selection”. Based on one or more arguments the program decides IF a specific situation is valid. IF so, THEN a specific part of the program is executed, ELSE another part of the program may be executed.

The expression to be evaluated can be very complex, consisting of multiple arguments in multiple constraints. We specify the constraints using Boolean algebra, resulting in complex expressions.

This pattern is applicable in:

- Any program language in any program that deals with complex Boolean expressions.
- Any structured design that uses some kind of pseudo code.
- Business or technical analyses where conditions are formulated.

### 2.2 Example

Regard the next well-known situation (at least, for pattern writers).

At EuroPLoP conferences not everyone is allowed to comment actively in workshops on pattern papers submitted to that conference. First of all, those people have to be present because review through mail or phone is not allowed. Besides that, only people in the same workshop stream are allowed to participate. But then again, only under restriction that they themselves are author of a pattern paper that's in a workshop in the same stream, unless the authors in the workshop agree that non-authors are allowed to participate too. Yet another restriction is applicable: only authors or non-authors that attend all workshops in this particular stream of workshops participate, unless other participants grant permission to come aboard. Finally, the author of the paper himself is not allowed to comment.

How would the resulting expression look like, to find out whether someone is allowed to comment in a particular workshop?

The resulting Boolean expression would look like (in “pseudo code”).

```

# Are you allowed to comment on the paper?
Answer by default = NO
IF physically present
THEN
    IF author of paper
    THEN
        IF author himself
        THEN
            Answer = NO
        ELSE
            IF attendee in all workshops up till now
            THEN
                Answer = YES
            ELSE
                IF permission to come aboard
                THEN
                    Answer = YES
                END
            END
        END
    ELSE
        IF non-authors allowed to comment
        THEN
            IF attendee in all workshops up till now
            THEN
                Answer = YES
            ELSE
                IF permission to come aboard
                THEN
                    Answer = YES
                END
            END
        END
    END
END
END

```

## 2.3 Problem

In complex expressions, consisting of multiple or nested statements using Boolean expressions, humans tend to lose comprehension and oversight of the expression. So there's the problem. Is there a way to reduce complexity and improve understanding?

## 2.4 Forces

People are easily seduced to increase complexity in order to show their intelligence. Likewise, others are unwilling to admit that they do not understand this complexity. They pretend they do.

In reasoning, a human being isn't a computer. It's easy to make mistakes in writing or evaluating these statements.

People strive for statement reductions; this is linked to showing intelligence. People do not use brackets unless necessary, since they know about priorities in handling the Boolean operators. They forget that others will not have the same knowledge.

People forget that these statements have to be maintained.

In real software programs usually some other code is added within the IF-statements, retrieving or manipulating data, reports, etc. This results in lengthy IF-statements spread over multiple pages, resulting in expressions and programs even more difficult to comprehend.

In nested statements some parts of the nested expressions have to be repeated in several branches of the expression. This is difficult to maintain and leads to problems.

## 2.5 Solution

The smart use of binary algebra allows us to reduce complexity. Instead of combining and nesting expressions we transform the complex expression into single statements. In this way a complex statement is reduced to a set of single statements that will not be nested. This is the first major part of the solution, reducing complexity and improving insight.

Unfortunately, it seems that we're no longer able to evaluate the complex expression. Fortunately, that only seems to be the case! By giving a well-chosen value to the result of each single statement, we are able to base the answer to the overall complex expression on the sum of these values. This is the second major part of the solution.

This well-chosen value is determined as follows.

- A positive answer to the 1<sup>st</sup> statement is rewarded with 1 "point".
- A positive answer to the 2<sup>nd</sup> statement is rewarded with 2 points.
- A positive answer to the 3<sup>rd</sup> statement is rewarded with 4 points.
- A positive answer to the 4<sup>th</sup> statement is rewarded with 8 points.
- Etcetera.

A negative answer to any statement is rewarded 0 points.

In general: the number of points to reward is a power of 2:  $2^{(\text{number of statement} - 1)}$ .

The sum of all points represents a unique combination of the arguments. The final statement is reduced to questioning the value of this sum.

## 2.6 Resolved example

In the given example the complex expression consists of following simple statements:

1. Attendee in all workshops
2. Non-authors allowed to comment
3. Permission to come aboard
4. Author himself
5. Author of paper
6. Physically present

Please pay attention to the order of the statements; it's one of the implementation issues. The table provides explanation and insight in the resulting values. Each column represents the answer to a single statement. The Value column represents the value of the sum. The Answer column represents the answer to the overall expression; this is the column that provides the basis for the simplified Boolean expression.

```
SUM = 0
```

```
IF Attendee in all workshops THEN Add 1 to SUM.
```

```
IF Non-authors allowed to comment THEN Add 2 to SUM.
```

```
IF Permission to come aboard THEN Add 4 to SUM.
```

```
IF Author himself THEN Add 8 to SUM.
```

```
IF Author of paper THEN Add 16 to SUM.
```

```
IF physically present THEN Add 32 to SUM.
```

```
IF SUM = 37 OR 38 OR 39 OR 45 OR 46 OR 47 OR 49 OR 50 OR 51  
OR 53 OR 54 OR 55
```

```
THEN
```

```
    Answer = YES
```

```
ELSE
```

```
    Answer = NO
```

```
END
```

\*\*\* Explaining table for above statements and expression

Sum	Phys. Present	Author	Author Himself	Come Aboard	Non-author	In all work.	Answer
0	N	N	N	N	N	N	N
1	N	N	N	N	N	Y	N
2	N	N	N	N	Y	N	N
3	N	N	N	N	Y	Y	N
4	N	N	N	Y	N	N	N
5	N	N	N	Y	N	Y	N
6	N	N	N	Y	Y	N	N
7	N	N	N	Y	Y	Y	N
8	N	N	Y	N	N	N	N
9	N	N	Y	N	N	Y	N
10	N	N	Y	N	Y	N	N
11	N	N	Y	N	Y	Y	N
12	N	N	Y	Y	N	N	N

Sum	Phys. Present	Author	Author Himself	Come Aboard	Non-author	In all work.	Answer
13	N	N	Y	Y	N	Y	N
14	N	N	Y	Y	Y	N	N
15	N	N	Y	Y	Y	Y	N
16	N	Y	N	N	N	N	N
17	N	Y	N	N	N	Y	N
18	N	Y	N	N	Y	N	N
19	N	Y	N	N	Y	Y	N
20	N	Y	N	Y	N	N	N
21	N	Y	N	Y	N	Y	N
22	N	Y	N	Y	Y	N	N
23	N	Y	N	Y	Y	Y	N
24	N	Y	Y	N	N	N	N
25	N	Y	Y	N	N	Y	N
26	N	Y	Y	N	Y	N	N
27	N	Y	Y	N	Y	Y	N
28	N	Y	Y	Y	N	N	N
29	N	Y	Y	Y	N	Y	N
30	N	Y	Y	Y	Y	N	N
31	N	Y	Y	Y	Y	Y	N
32	Y	N	N	N	N	N	N
33	Y	N	N	N	N	Y	N
34	Y	N	N	N	Y	N	N
35	Y	N	N	N	Y	Y	N
36	Y	N	N	Y	N	N	N
37	Y	N	N	Y	N	Y	Y
38	Y	N	N	Y	Y	N	Y
39	Y	N	N	Y	Y	Y	Y
40	Y	N	Y	N	N	N	N
41	Y	N	Y	N	N	Y	N
42	Y	N	Y	N	Y	N	N
43	Y	N	Y	N	Y	Y	N
44	Y	N	Y	Y	N	N	N
45	Y	N	Y	Y	N	Y	Y
46	Y	N	Y	Y	Y	N	Y
47	Y	N	Y	Y	Y	Y	Y
48	Y	Y	N	N	N	N	N
49	Y	Y	N	N	N	Y	Y
50	Y	Y	N	N	Y	N	Y
51	Y	Y	N	N	Y	Y	Y
52	Y	Y	N	Y	N	N	N
53	Y	Y	N	Y	N	Y	Y
54	Y	Y	N	Y	Y	N	Y
55	Y	Y	N	Y	Y	Y	Y
56	Y	Y	Y	N	N	N	N
57	Y	Y	Y	N	N	Y	N
58	Y	Y	Y	N	Y	N	N

Sum	Phys. Present	Author	Author Himself	Come Aboard	Non-author	In all work.	Answer
59	Y	Y	Y	N	Y	Y	N
60	Y	Y	Y	Y	N	N	N
61	Y	Y	Y	Y	N	Y	N
62	Y	Y	Y	Y	Y	N	N
63	Y	Y	Y	Y	Y	Y	N

## 2.7 Consequences

Benefits:

- In general, the complexity in complex statements reduces drastically, with major benefits.
  - Resulting statement provides much more insight and understanding to everyone involved, including business analysts, designers and programmers, even managers.
  - Programming errors are much easier to detect and correct.
  - Changes in criteria (maintenance) are much easier to program.
  - Performance in computing the statement improves.
  - Takes less lines of code.
- By isolating the individual conditions, the analysis of “non structured business talk” (like the example) is structured, thus improving understanding and insight (even to the business themselves).

Pitfalls:

- The “not invented here syndrome” makes people stick to their own way of working.
- It takes some intellectual effort to get the point of this pattern. If people don’t understand the way of working at first glance, they tend to lay the paper aside.
- Inconsistency of the explaining table and the statements is a risk, especially since a lot of people don’t like or forget to add comments to their code.
- As my shepherd reminded me: although the pattern does reduce complexity, the complexity does not go away!

## 2.8 Implementation Issues

Think carefully about numbering the single statements, since doing so can optimise the resulting statement. It is recommended to reward the most selective single statement with the highest number: this will enable the use of the mathematical  $>$  or  $<$  instead of  $=$ .

It is very wise to include comment in the design or the program explaining the algorithm. Especially including the table.

However, the length of the explaining table doubles with each condition added. Of course, this leads to a situation difficult to survey or understand. Therefore you should use the accompanying tool (in Excel) for situations of let’s say more than six conditions. The spreadsheet helps you in situations of up to 10 conditions. If you have statements where you

need more than 10 conditions you should really rethink this expression because it will probably never result in manageable code.

Be aware that there might be a performance penalty in applying this pattern, since you might not be able to take advantage of some optimising statements programming languages provide.

Start the explaining table with the last statement (with the most points) at the left.

The thinking is in the table. However, this is very much fit for reuse. In fact, I've done it for you for up to 5 single statements, apart from the column "Answer" 😊.

If you change the table, make sure to check that exactly the same number of "Y" and "N" appear in each column.

## 2.9 Introduction to the spreadsheet

This pattern is accompanied by a spreadsheet (using Microsoft Excel) that is helpful in applying this pattern. In the middle of the spreadsheet you will find a cell explaining how the spreadsheet is constructed.

Using the spreadsheet is quite simple:

- Specify the conditions you want to evaluate; the spreadsheet will calculate the "well-chosen" value to be used.
- Specify any combination of these conditions; the spreadsheet will calculate the sum of all conditions evaluated.
- Specify if the resulting answer of this condition should be either Yes or No; the spreadsheet will provide the accompanying value that should be evaluated in the resulting expression.

No doubt the spreadsheet can be enhanced with fancy features and extensive explanation. If you do so, please share your work with the pattern community.

## 2.10 Known Uses

The author used this pattern 10 years ago (when he was still a program designer) in several programs. As far as he knows this program still uses this solution.

Recently, I discovered another way of using this pattern, although some people might even argue that it is even another pattern. I was facing the situation of about 40 combinations of 13 different parameters, trying to find identical instances. By using this pattern I was able to find all identical instances within a few minutes, just by sorting the resulting spreadsheet on the sum of all well-chosen values of each of the 13 different parameters.