Patterns for Selection
Version 2

Joseph Bergin

This paper presents a simple pattern language (as a pattern, actually), to use in writing code that requires selecting from alternative actions.

There are four kinds of patterns here: Selection patterns proper, Strategy patterns, Auxiliary patterns, and some Stylistic patterns.

The Selection patterns are:
- Whether or Not
- Alternative Action
- Range of Possibility
- Sequential Choice
- Unrelated Choice
- Independent Choice

The strategy patterns are
- Short Case First
- Default Case First

The auxiliary patterns are:
- Positive Condition
- Function for Complex Condition
- Function for Complex Action

You are at a point in a program at which one of two or more different actions must be performed. You must select between the alternatives.

Whether or Not

(Also known as: Guarded Action, Guarded Command).

You are in a situation in which some action may be appropriate or inappropriate depending on some testable condition.

For example: in a power plant simulation, it may be necessary to shut down a generator when it overheats.

```
if (measuredHeat() > subBoilThreshold)
{   shutDownGenerator();
}
```

You don't need to repeat the action, only to decide whether or not it should be done. There are no other actions to do (only) when this one is not done. You want to write simply understood code.

Therefore, use an IF statement without an ELSE part, expressing the test condition positively if possible.

IF <condition>
    <action>

Note that in most languages there is an if statement intended for precisely this situation.

If, instead, you need to repeat the action, see Patterns for Loops.

Alternative Action

You are in a situation in which one of exactly two actions is appropriate depending on some testable condition.

When the condition holds you want to do one action, and when it does not hold you want to do some different action. There are exactly two actions and exactly one condition, which may be true or false.

For example a student may pass or fail an exam depending on the value of the numeric grade.

```
if ( numericGrade > 60 )
{   output (“passing”);
}
else
{   output(“failing”);
}
```

(due to Rick Mercer)

Therefore, use a single IF statement with an ELSE part, expressing the test in positive terms.

```
IF <condition>
        <one action>
ELSE
        <another action>
```

If you try to apply Whether or Not (twice) to this case you will find yourself needing to write the negation of the condition.

```
if ( numericGrade > 60 )
{   output (“passing”);
}
if ( numericGrade <= 60)
{   output(“failing”);
}
```

This is both wasteful of computer time and very error prone. Also, if problem changes a bit in the future and you change one of the conditions, it is easy to forget to change the other. Alternative Action makes it unnecessary to repeat the condition for the else part. See Unrelated Choice.

Positive Condition

You are applying Alternative Action and are wondering how to define the condition and lay out the IF-ELSE statement.

Most people can more effectively read a positive statement than a negative one. You want your code to be as readable as possible.

For example, suppose you have a robot simulation in which the robot must move but must also contend with obstructions in the path. Suppose you have a boolean test as a primitive in a robot language:

```
boolean frontIsBlocked();
```

Suppose that you want to move if possible, but turn Left instead if it is impossible to move forward. The following are equivalent:

```
if( frontIsBlocked())
    turnLeft();
else
    move();

if ( ! frontIsBlocked())
    move();
else
    turnLeft();
```

The first version is more readable and is preferred. It expresses a positive condition.

Therefore, when writing conditions, express them positively whenever possible.

Function for Complex Condition

You are applying Whether or Not or Alternative Action and trying to write the condition. You realize that the condition is complex.

Most people find it very difficult to read, understand, and remember complex Boolean conditions.

It may be desirable to write a function reversing the logical sense of a given test to apply Express Positive Condition in all circumstances. For example, in the above example, one could write the function

```
boolean frontIsClear() { return ! frontIsBlocked(); }
```

and use this in place of ! frontIsBlocked(). This makes it possible to Express Positive Condition even when using other selection patterns than Alternative Action.

Therefore, write a Boolean function to capture the condition and call this function in the if or if-else statement. Functions that return Boolean values are normally called Predicates.

The name of the function should be easy to remember, should exactly express the meaning of the condition, without expressing its details, and should do so in a positive way.

Note that the name chosen in the above, frontIsClear, is itself expressed positively. This is preferred over the equivalent frontIsNotBlocked.

Function for Complex Action

You are applying Whether or Not or Alternative Action and trying to lay out the IF statement. You realize that one or more of the actions is complex.

Most people find that reading a complex action within an if distracts from the overall flow of understanding of the program. This is because they need both the detail of the action, to understand what it does, as well as the general idea of the action, in order to understand the larger program that contains it.

Therefore, write a function to encapsulate the complex action(s).

This is especially effective if you can choose a good name for the action that captures exactly the nature of the complex action. This should then become the name of the function.

```
if ( nextToABeeper())
    pickBeeper();
else
{   …
    // hundreds of statements to find the beeper
    …
}
```

can be re cast as:

```
if ( nextToABeeper())
    pickBeeper();
else
    findBeeper();
```

Short Case First

You are applying Alternative Action and trying to lay out the IF-ELSE statement. One of the actions can be expressed simply in a statement or two. One is much longer. For some reason it is not desirable to apply Function for Complex Action.

You want your reader to be able to read and understand the code as simply as possible. You also want the reader to be able to easily determine if this is an if with an else or without an else.

Therefore, arrange the code so that the short case is written as the if (not the else) part.

```
if (someCondition())
    // aStatement
else
{   ...
    // lots of statements
    ...
}
```

This will permit the reader to easily dispense with one case before forgetting the condition that is used to choose between cases.

You may need to use Function for Complex Condition to enable this pattern.

The next patterns Range of Possibility and Sequential Choice presented below in this language are very special purpose and they are easy to abuse. Most often the better design is to use polymorphism in an object-oriented class hierarchy to do the kind of choosing of alternatives that these do. A polymorphic design will be much easier to extend and maintain. The following techniques are needed in older programming languages, but seldom in well written object-oriented programs.

Range of Possibility

You are thinking of applying something like Alternative Action, but you have more than two possibilities. You must choose exactly one of several actions to perform based on some condition.

You also have a situation in which the choices to be made depend on the current value of a computed expression (perhaps a single variable) and that expression has a discrete (integer) type.

You want to test the value of this expression exactly once for economy of both execution and reading and you want to avoid inconsistency errors in the future if the problem changes.

Therefore, use a switch statement with the expression as the test value and the various values of that expression as the case labels.

```
switch (skyColor)
{   default: output("Don't know where we are."); break;
    case Color.red: output("This must be Mars."); break;
    case Color.blue: output("This must be Earth."); break;
    case Color.green: output("This must be Moldy."); break;
}
```

Each case should end with break. This includes the last case. The problem may need modification and new cases may arise. It is easiest and safest if new cases can be added without modifying existing cases.

If a break after a case is not appropriate, so that execution should continue through to the next case, be sure to document that fact. Thus several values of the tested expression can result in the same action.

```
switch (skyColor)
{   default: output("Don't know where we are."); break;
    case Color.red:  // No break. Continuing.
    case Color.blue: output("This must be the Sol system."); break;
    case Color.green: output("This must be Moldy."); break;
}
```

Be sure to include a break after the last case. Chances are that the switch will be extended in the future.

Default Case First

You are applying Range of Possibility.

The code should be readable and easily understood. It is easy to forget the default case. The reader may want to know what happens if none of the case labels matches the computed value of the test expression.

Therefore, consider writing the default case first so that it may be easily seen and considered. This was shown in the above examples. Be sure to give the default case a break, no matter where it appears.

Sequential Choice

(Also known as: elsif, one of many.)

You are in a situation in which you need to choose exactly one of several possible actions, but which action does not depend on the value of a single expression. Instead, suppose each action depends on a separate testable condition.

You want the code layout to be pleasing to both the eye and the mind. You want a structure that is easy to read and understand. Each action is guarded by its own condition, and after you find one condition true you want to execute its associated action and at that point you want to finish.

Therefore, write a sequence of IF's, where each IF but the last has an ELSE part that consists entirely of another IF.

```
int participants = myParty.size();
if (participants > 15000)
{   rentTheSuperdome();
}
else if (participants > 1500)
{   rentTheCivicCenter();
}
else if (participants > 150)
{   rentATent();
}
else
{   rentAMovie(); // default case, no party at all.
}
```

The formatting, with else and if on the same line, makes it clear that this is a Sequential Choice and not a sequence of Whether or Not applications. Do not indent the subsequent else parts. Some languages have a special keyword (elsif) to handle this case.

Unrelated Choice

You are in a situation in which you have many actions and many conditions. You may want to execute several of the actions if their associated conditions are true.

The key here is that you may want to execute more than one of the actions and each action has a condition that determines if it should be executed.

Don't get confused by the similarity to the above patterns. This is most likely just a set of Guarded Actions that you need to apply in some order. The order may be arbitrary or not, depending on the specific situation.

```
if (roofIsLeaking())
{  callRoofer();
}
if (sinkIsLeaking())
{  callPlumber();
}
if (floorIsLeaking())
{  callExcavator();
}
```

Independent Choice

You are in a situation in which exactly one action must be chosen, but which action depends on several factors, not just a single one.

If the factors to be considered are independent and there are only a few such factors (three or less) then nested IF statements may be an adequate solution. Two factors that are independent of each other provide for four possibilities when taken together. For example if the factors are (rectangle or not) and (has background or not) then we get the four possibilities shown in Figure 1. Three independent factors will similarly result in eight possibilities.

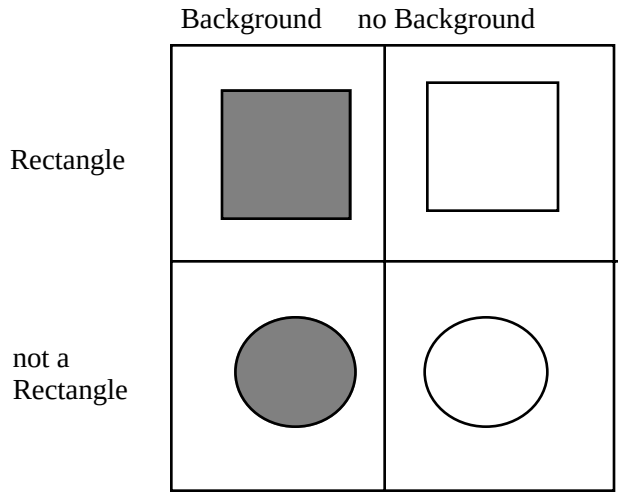

Figure 1. Two independent choices

Another example is one in which you have two variables and you must decide on one action depending on whether each of these variables is negative or not. The following nested IF structure will work in this case. Here we have chosen to consider the x variable first and then the y variable.

```
if(x >= 0)
{   if(y >= 0)
    {   System.out.println("First quadrant");
    }
    else
    {   System.out.println("Second quadrant");
    }
}
else // x < 0
{   if(y >= 0)
    {   System.out.println("Third quadrant");
    }
    else
    {   System.out.println("Fourth quadrant");
    }
}
```

Note that the structure of the inner IF is repeated in both parts of the outer IF , though the actions are not the same.

However, if we reverse the inner and outer structures we must arrange the innermost actions differently.

```
if(y >= 0)
{   if(x >= 0)
    {   System.out.println("First quadrant");
    }
    else
    {   System.out.println("Third quadrant");
    }
}
else
{   if(x >= 0)
    {   System.out.println("Second quadrant");
    }
    else
    {   System.out.println("Fourth quadrant");
    }
}
```

If you have three actions instead of four, but the choice depends on two independent factors, choose the inner and outer conditions so that one of the inner if's applies the same action in both cases. Then you can omit one inner test and just apply that action. For example, suppose you are running a power plant simulation and how much power output you get from the reactor depends on two factors, the state of the transmission lines and the state of the reactor itself. The following are equivalent.

```
if(transmissionOK())
{   if(reactorOK())
    {   fullPower();
    }
    else
    {   reducedPower();
    }
}
else
{   if(reactorOK())
    {   fullPower();
    }
```

```
    else
    {   shutDown();
    }
}

if(reactorOK())
{   if(transmissionOk()
{   fullPower();
{
else
{   reducedPower();
}
}
else
{   shutDown();
}
```

The second form has simpler structure. Of course, you may then want to write the Short Case First.

Often it is better to apply Function for Complex Action instead of nesting structures. To apply that pattern to Independent Choice, let each inner IF be represented by a separate function call. You will need to write two functions to apply this: one for the IF part of what is here the outer IF and another for the ELSE part.

Sometimes you have several things to consider but only two possible actions. In this case you need a compound expression to test. Apply the pattern Use Function for Complex Condition. Write a function that returns true when one of the required combinations of conditions is met and call this function as the test in the IF. In the power plant situation, there may be only two choices of action, full power and shut down. In this case some predicate will let you choose between them, though it may represent a complex condition.

Note that it may be more readable to use compound conditions with *and* and *or* operators and avoid nesting. For example, the quadrant problem can be solved using Unrelated Choice as

```
if(x >= 0 && y >= 0)
{   System.out.println("First quadrant");
}
if(x >= 0 && y < 0)
{   System.out.println("Second quadrant");
}
if(x < 0 && y >= 0)
{   System.out.println("Third quadrant");
}
if(x < 0 && y < 0)
{   System.out.println("Fourth quadrant");
}
```

However, you may then want to apply Function for Complex Condition. This also requires executing tests done previously. If the problem changes this entire structure needs to be reanalyzed as a whole.

Stylistic Patterns

The four patterns in this group can help you make your code more readable, and they can also help you avoid problems in the future when a program must be modified and updated.

One Liner

You are writing a selection structure and notice that all of the parts are short.

It is a good idea to use the horizontal as well as the vertical "real estate" of your page, since if you can fit more on a page, the reader will need to turn fewer pages to follow your code. Your reader also expects that what is on one line all goes together.

Therefore, if the entire structure fits comfortably on one line, then put it on one line.

```
if ( numericGrade > 60 ) { output (“passing”);} else { output(“failing”);}
```

Brace All

You are writing a selection or other structure and notice that some of the actions consist of single statements. The language doesn't require that you write braces or other grouping symbols in this situation.

However, you recognize that programs change as the problems that they solve change. In real programming this is a very frequent occurrence. If you have a single statement in an action, chances are that later it may need more statements.

Therefore, completely brace all statement parts in all structures when you first write the program.

```
if (measuredHeat() > subBoilThreshold)
{   shutDownGenerator();
}
```

can be modified more easily and with less possibility for error than the logically equivalent

```
if (measuredHeat() > subBoilThreshold)
    shutDownGenerator();
```

Braces Line Up

You are writing a structured statement that requires braces or other grouping symbols. You want your code to be as readable as possible.

When structures are nested, the indentation structure is often hard to follow. It is especially hard when the inner structures end and the outer structure resumes. The eye cannot always easily see what goes with what level of the overall structure. This is one reason for Function for Complex Action, of course.

The braces of a structure give its real intent, independent of how it is indented.

Therefore, when writing brace symbols or other grouping symbols such as parentheses, if the opening and closing symbol don't both fit on the same line, then make them line up exactly vertically.

This stylistic pattern has been followed throughout this paper. It is somewhat different from the style seen in most C++ and Java books, however. The more typical style would have the opening brace at the end of the line on which it opens and the closing brace under the keyword that indicates the structure.

```
if (measuredHeat() > subBoilThreshold) {
    shutDownGenerator();
}
```

Note that when the opening brace begins a line, you can put a full statement on that line as well, so that you don't waste vertical real estate.

Indent for Structure

You are writing a structured statement using these (or other) patterns. You want to write readable code. In particular you want to indicate to your reader what the individual parts of your structure are.

The eye is good at grouping things. It is probably better at this than the mind.

Therefore, the parts of a structure should be indented from the keywords and punctuation symbols that define its structure. All of the statements at the same level of the structure should be indented exactly the same amount.

Don't indent too much or you waste horizontal real estate. Don't indent too little, or the eye won't see the structure. See the code fragments above for examples of the use of this pattern.

Note. These patterns were informally discussed in a workshop at SIGCSE '99 in New Orleans. The participants were Eugene Wallingford, Owen Astrachan, Rick Mercer, Robert Duvall, Alyce Brady, Viera Proulx, Richard Rasala, and Kathy Larson. I thank them for the many improvements they suggested and also for their support.