

Patterns for Selection
Version 6

Joseph Bergin
Pace University
New York, NY USA
jbergin@pace.edu

This paper presents a simple pattern language to use in writing code that requires selecting from alternative actions.

> A group of educators first met at ChiliPlop '98 to discuss ways that we might use patterns to instruct novices. Since patterns try to capture best practice, and since novices need this kind of information also, it seemed like a natural thing to try. A continuing working group has grown out of our meetings that week and this paper is one of the results. It is intended for the first introduction to the topic for novices just learning to program for the first time. Everything here should be known and probably obvious to a working programmer, but to a novice this is new and possibly intimidating. This work is intended for educators and for the students that they teach.
>
> Just as with more sophisticated patterns, we hope that the pattern format will get the reader to analyze what they do in terms of context and forces. You can learn more about the Elementary Patterns group at: http://www.cs.uni.edu/~wallingf/patterns/elementary

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

The Selection patterns are as follows. These choose different kinds of selection structures.

Whether
Alternative Action
Return Not Else
Conditional Expression
Range of Possibility
Sequential Choice
Unrelated Choice
Independent Choice
Partial Dependence

The strategy patterns are as follows. These help you design a selection.

Short Case First
Default Case First

The auxiliary patterns are as follows. These give general advice about simplifying the structure of your programs.

Positive Condition
Function for Complex Condition
Function for Complex Action

The stylistic patterns are as follows. These are concerned with the readability and maintainability of your programs.

One Liner
Brace All
Braces Line Up
Indent for Structure

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 (Selection)

(Also known as: Guarded Action, Guarded Command, Optional Action, Whether or Not.)

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 it should be done or not. There are no other actions to do instead of this one. You want to write simply understood code.

Therefore, use an IF statement without an ELSE part, expressing the test as a *Positive Condition* if possible.

IF <condition>
    <action>

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

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

Note: Conditions are sometimes called guards. The condition guards the command and only permits it to be executed when the guard is true.

Alternative Action (Selection)

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.

Therefore, use a single IF statement with an ELSE part, expressing the test as a *Positive Condition*.

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

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”);
}
```

If you try to apply *Whether* (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. If the 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*.

Conditional Expression (Selection)

You are writing a program and need to compute a value to use in an expression or an assignment. Which value to use depends on some single condition.

An *Alternative Action* with a temporary variable can do the job, but it is awkward. For example

```
int temp;
if(emp.dueBonus())
{   temp = dept.standardBonus();
}
else
{   temp = 0;
}
salary = emp.basicSalary() + temp;
```

This is wasteful of space and overly verbose. It also does not express to the reader that the intent is to compute a value that depends on the condition.

Therefore consider using the conditional operator instead.

```
salary = emp.basicSalary() + emp.dueBonus() ? dept.standardBonus() : 0 ;
```

However, if either the condition or one of the two values requires a long expression, it may be better to use *Alternative Action*. This can only be used when there are exactly two alternatives and normally when the alternatives are simple expressions. As a matter of style, it might be helpful to parenthesize the conditional expression and restrict your use of this pattern to *One Liners*.

```
salary = emp.basicSalary() + (emp.dueBonus() ? dept.standardBonus() : 0) ;
```

Don't use this at all unless the results are very readable.

Positive Condition (Auxiliary)

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 (Auxiliary)

You are applying *Whether* 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.

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.

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

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

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

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

Sometimes you have several things to consider but only one or two possible actions. In this case you need a compound expression to test. Apply the pattern *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 a power plant simulation, 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.

```
If(reactorError() || (transmissionError() && heatRising()))
{   stutdownReactor();
}
else
{   fullPower();
}
```

If we write a function to capture the condition, this becomes

```
Boolean shutdownRequired()
{   return reactorError() || (transmissionError() && heatRising());
}
```

```
…
If(shutdownRequired ()){ shutdownReactor();} else{ fullPower();}
```

An added benefit here is that if the condition for shutdown changes, it is easy to find the place to change the code. The code is also reusable elsewhere in the program and it has one point of change if the problem changes.

Function for Complex Action (Auxiliary)

You are applying *Whether* 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 (Strategy)

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. Also, it is very important that you express *Positive Conditions*.

If for some reason you must have a long first case, then it is useful to comment the else with the negation of the tested condition. This need not use programming syntax, but can explain in ordinary language what must be true here. (Should this be a pattern (Comment Else Part)?)

```
if (nextToABeeper())
{...
    // lots of statements
    ...
}
else // NOT next to a beeper.
{   ...
    // lots of statements
    ...
}
```

Polymorphism First

The selection patterns *Range of Possibility* and *Sequential Choice* presented below in this language are 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. However, a full discussion of this is beyond the scope of this paper. The following techniques are needed in older programming languages, but seldom in well written object-oriented programs.

Range of Possibility (Selection)

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.

An example which illustrates this point as well as applying the *Function for Complex Condition* pattern follows:

```
// assume traditional month numbering from 1 to 12
int daysInMonth(int month, int year)
{   int result = 0;
    switch (month)
    {   case 2:{    if (isLeapYear(year))
                    {   result = 29;
                    }
                    else
                    {   result = 28;
                    }
                    break;
               }

        case 4:           // no break
        case 6:           // no break
        case 9:           // no break
        case 11:  {result = 30; break;}

        default:  {result = 31; break;}
    }
```

```
}
```

Note that polymorphic method dispatch may be a better solution than this for many problems.

Default Case First (Strategy)

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 immediately 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. Be sure to give the default case a break, no matter where it appears. Repeating an earlier example:

```
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;}
}
```

Sequential Choice (Selection)

(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* applications. Do not indent the subsequent else parts or it will look like *Independent Choice* and you will also give up valuable horizontal real estate. Some languages have a special keyword (elsif) to handle this case.

Note that polymorphic method dispatch may be a better solution than this for many problems.

Return Not Else (Selection)

You are writing a function. The value you want to return depends on some set of conditions.

It is good to be able to dispense with things as you read them rather than having to remember them as you read further.

Therefore, use if statements with return statements, rather than with else parts.

The following code fragment are equivalent.

```
boolean isLeapYear(int year)
{   boolean result = false;

    if (year % 400 == 0)
    {   result = true;
    }
    else if (year % 100 == 0)
    {   result = false;
    }
    else if (year % 4 == 0)
    {   result = true;
    }
    return result;
}

boolean isLeapYear(int year)
{   if (year % 400 == 0) {return true;}

    if (year % 100 == 0) {return false;}

    if (year % 4 == 0) {return true;}

    return false;
}
```

The second form lets the reader know that the final result in each case has been computed and won't be altered in what follows. This is because if the return is executed, the function immediately terminates, returning the specified value.

A second example re writes the daysInMonth function and applies the same idea to *Range Of Possibility*.

```
// assume traditional month numbering from 1 to 12
int daysInMonth(int month, int year)
{   switch (month)
    {   case 2:{   if (isLeapYear(year))
                   {   return 29;
                   }
                   return 28;
               }
        case 4:           // no break
        case 6:           // no break
        case 9:           // no break
        case 11:  {return 30;}

        default:  {return 31;}
    }
}
```

Note that break statements would be redundant here. This is a variation on *Return Not Else*.

Unrelated Choice (Selection)

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.

What sets this problem apart 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.

Therefore, just apply *Whether* for each condition/action pair.

Don't confuse this with *Sequential Choice*. The difference is that here we may need to execute several of the actions, not just one. 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 (Selection)

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.

Therefore, when you have a small number of independent conditions used to choose exactly one action, use nested if statements, where each level of nesting controls one condition.

For example if the factors are (rectangle or not) and (filled or not) then we get the four possibilities shown in Figure 1. Three independent factors will similarly result in eight possibilities.

```
if (a.isRectangle())
{   if(a.isFilled())
    {   hatchBox();
    }
    else
    {   openBox();
    }
}
else // not rectangle
{   if(a.isFilled ())
    {   EasterEgg();
    }
    else
    {   justABall();
    }
}
```

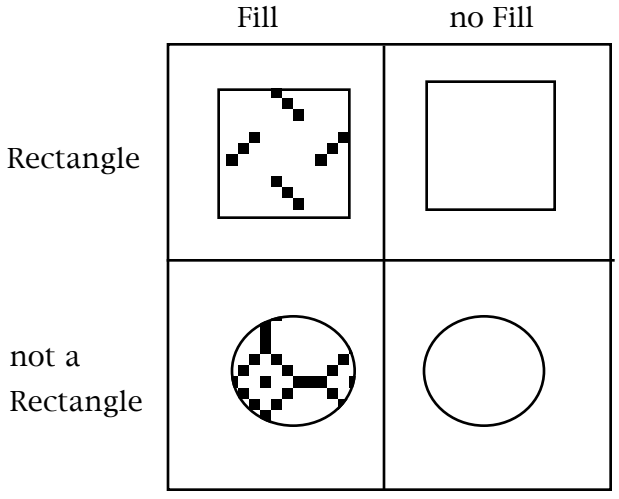

Figure 1. Two independent choices

However, nested structures can be very difficult to read. One can usually apply *Function for Complex Action* to avoid this pattern.

Partial Dependence

Sometimes you have multiple conditions and need to choose only one action of several, but the same action needs to be executed for more than one combination of conditions. In this case it is often helpful to draw a picture like Figure 2 in which the conditions label the divisions of the rectangle, and the resulting inner boxes hold the actions.

For example, suppose that we need to consider both the current state of the power reactor and the condition of the transmission lines. We might have the rules as shown in Figure 2.

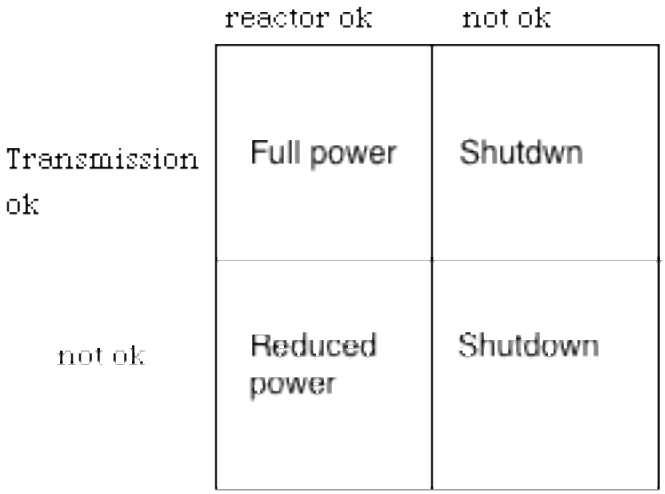

Figure 2. Partial Dependence

The following are logically equivalent.

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

if(reactorOK())
{   if(transmissionOk()
    {   fullPower();
```

```
    }
    else
    {   reducedPower();
    }
}
else
{   shutDown();
}
```

The second form has simpler structure.

Therefore, 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. In particular if a column in the picture has the same action in each box, make the column condition the outer one.

Of course, you may then want to write the *Short Case First*:

```
boolean reactorError(){ return ! reactorOK(); }
…
if(reactorError())
{   shutdown();
}
else
{   if(transmissionOk())
    {   fullPower();
    }
    else
    {   reducedPower();
    }
}
```

This then maps easily to a *Sequential Choice*, since the outer else has just an if statement and nothing more.

```
if(reactorError())
{   shutdown();
}
elseif(transmissionOk())
{   fullPower();
}
else
{   reducedPower();
}
```

But if you have come this far then you might as well write the transmissionError function and have, finally:

```
if(reactorError())
{   shutdown();
}
elseif(transmissionError())
{   reducedPower();
}
else
{   fullPower();
}
```

This last version orders the actions from most critical to least critical and it has symmetric conditions, both being stated in terms of possible errors.

The next four patterns 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 (Stylistic)

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 logically.

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 (Stylistic)

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 (Stylistic)

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 (Stylistic)

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 is.

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. Also compare the following.

```
if(reactorOK()) // Too much. Losing real estate fast.
{          if(transmissionOk()
           {          fullPower();
           }
           else
           {          reducedPower();
           }
```

```
}
else
{          shutDown();
}

if(reactorOK()) // Too little. Eye can't line up if sections are long.
{ if(transmissionOk()
  { fullPower();
  }
  else
  { reducedPower();
  }
}
else
{ shutDown();
}

if(reactorOK()) // Just right.
{   if(transmissionOk()
    {   fullPower();
    }
    else
    {   reducedPower();
    }
}
else
{   shutDown();
}
```

Conclusion

We will close with some examples that show again how these patterns might work together and also give some additional ideas.

An example using *Independent Choice* arises if 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("Fourth quadrant");
        }
    }
    else // x < 0
    {   if(y >= 0)
        {   System.out.println("Second quadrant");
        }
        else
        {   System.out.println("Third 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("Second quadrant");
        }
    }
    else
    {   if(x >= 0)
        {   System.out.println("Fourth quadrant");
        }
        else
        {   System.out.println("Third quadrant");
        }
    }
```

In situations like this 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. For example, our graphic example

```
if (a.isRectangle())
{   if(a.isFilled())
    {   hatchBox();
    }
    else
    {   openBox();
    }
}
else // not rectangle
{   if(a. isFilled ())
    {   EasterEgg();
    }
    else
    {   justABall();
    }
}
```

becomes

```
void handleRectangle(Figure a)
{   if(a.isFilled())
    {   hatchBox();
    }
    else
    {   openBox();
    }
}

void handleOval(Figure a)
{   if(a. isFilled ())
    {   EasterEgg();
    }
    else
    {   justABall();
```

```
    }
}
…
if (a.isRectangle())
{   handleRectangle(a);
}
else // not rectangle
{   handleOval();
}
```

Note that it may be more readable to use compound conditions with *and* and *or* operators and avoid nesting, using just *Whether*. 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("Fourth quadrant");
}
if(x < 0 && y >= 0)
{   System.out.println("Second quadrant");
}
if(x < 0 && y < 0)
{   System.out.println("Third quadrant");
}
```

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

Following is another example, taken from a compiler course project. The if statement does not convey the intent of either its condition or its body.

```
if  (!(expr instanceof ConstantExpression)
       && !(  expr instanceof VariableExpression
              &&  (   ((VariableExpression)expr).getScope() < 0
                  ||  (   ((VariableExpression)expr).getScope() == 0
                          && !((VariableExpression)expr).isIndirect()
                      )
                  )
           )
    ) // must copy to runstack
{   if(expr.getType().getSize() == 2)
    {   codegen.gen2Address(is, codegen.stackptr, codegen.immed, 0, 2);
        reg = codegen.loadReg(expr, currentlevel);
        codegen.gen2Address(st,reg,codegen.ireg, codegen.stackptr, 0);
        codegen.freeTemp(codegen.dreg,reg);
    }
    else
    {   size = expr.getType().getSize();
        codegen.gen2Address(is, codegen.stackptr, codegen.immed, 0, size);
        moveBlock(expr, codegen.ireg, codegen.stackptr, 0);
    }
}
```

If we apply *Function for Complex Condition* and *Function for Complex Action* to this we obtain the following. The names of the functions clearly convey the intent. The new predicate function is easy to follow since it returns as soon as it has sufficient information to do so.

```
boolean mustCopyToRunstack(Expression expr)
{   if (expr instanceof ConstantExpression) { return false; }
    // may assume it is a variable Expression
    VariableExpression temp = (VariableExpression) expr;
    int scope = temp.getScope();
    if (scope < 0 ) { return false; }
    boolean directValue = ! temp.isIndirect();
    if (scope == 0 && directValue) { return false; }
    return true;
}
```

We have used a local variable, scope, to avoid calling the getScope function twice and have used another local, directValue, in an attempt to increase comprehensibility of the final if statement in that predicate function.

```
void copyToRunStack(Expression expr)
{   if(expr.getType().getSize() == 2)
    {   codegen.gen2Address(is, codegen.stackptr, codegen.immed, 0, 2);
        reg = codegen.loadReg(expr, currentlevel);
        codegen.gen2Address(st,reg,codegen.ireg, codegen.stackptr, 0);
        codegen.freeTemp(codegen.dreg,reg);
    }
    else
    {   size = expr.getType().getSize();
        codegen.gen2Address(is, codegen.stackptr, codegen.immed, 0, size);
        moveBlock(expr, codegen.ireg, codegen.stackptr, 0);
    }
}
```

Using these two functions, the intent now becomes entirely clear. We simply need to determine *Whether* to copy the item to the run stack.

```
if(mustCopyToRunstack(expr))
{   copyToRunstack(expr);
}
```

Further Reading

I would like to refer the reader to three additional sources of information. First is the paper on *Patterns for Loops* by Astrachan and Wallingford, which can be found on the web at http://www.cs.duke.edu/~ola/patterns/plopd/loops.html. Next, "Smalltalk Best Practice Patterns", by Kent Beck (Prentice-Hall, 1996) wonderfully shows the power of simplicity in improving our programs. Finally, Richard Gabriel's *Simply Understood Code* pattern language precedes and extends the ideas expressed here. It can be found on the Wiki web at: http://c2.com/cgi/wiki?SimplyUnderstoodCode.

Acknowledgements

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.

It was formally discussed in a workshop at EuroPLoP '99 at Kloster Irsee in Germany. I also thank the participants for many suggestions and encouraging support.

The pattern shepherd is Robert Hanmer. He has also suggested many improvements in both style and content.

The example in *Alternative Action* is due to Rick Mercer. The days in the month example is from Richard Rasala, and the example in *Unrelated Choice* comes from Owen Astrachan. The final example, from the compiler, does actually appear in my own compiler course materials. It should be no surprise that I frequently get asked to explain it.