Deciding between if and switch when writing code (part 2 of 2)

Experiment performed at the 2009 ACCU Conference
First published in CVu vol. 21 no. ?

Derek M. Jones Knowledge Software Ltd derek@knosof.co.uk

1 Introduction

This article is the second of two that investigates the possible factors influencing developers in their choice of selection statement, i.e., deciding whether to use an if-statement or a switch-statement to implement some desired functionality. Two sources of data are analysed: the first article[2] analysed measurements of existing code and this second one discusses the results of an experiment carried out at the 2009 ACCU conference.

The source code measurements in the first article showed that the switch-statement rapidly becomes the selection statement of choice as the number conditionally executed statement sequences increases; see Figure .1. The 2009 ACCU experiment asked subjects to write various function definitions whose specification each involved behavior that depended on one variable that could take on various values.

Previous experience has shown that when asked to solve one simple problem developers often quickly fall into using a fixed pattern when answering. One of the aims of the ACCU experiments is for them to reflect actual work practices (to be ecologically valid is the technical terminology). In an attempt to prevent subjects following a pattern of answers that they would not follow in a work related environment each problem was split into two independent sub-problems.

1.1 The hypothesis

The experimental problem contained two independent subproblems and separate hypothesis derived for each subproblem.

1. Recognition of function call sequence order. Subjects are more likely to correctly recall the sequence of a previously seen sequence of function calls if the names of those functions follow a commonly occurring pattern, e.g., alphabetic/numeric order or having a (hopefully) recognizable order such as the sequence start-process-end. The details are discussed below.
2. **`if`**/**`switch`** statement choice. The decision on whether to use an **`if`** or **`switch`** statement is strongly effected by the number of conditional arms expected to appear in the final code. A secondary hypothesis is also tested:

    - Experienced developers will be aware that code written today frequently has to be modified in the near future because of changes to requirements or other parts of the code base. A developer's expectation of future changes to code that is being written now may also be a factor in the choice of selection statement.

      To test this hypothesis problems were designed to either involve quantities that subjects were thought likely to consider as being open-ended (living space related information was used e.g., trees in a garden) or to involve quantities thought likely to be considered to be closed by subjects (human body related information was used e.g., number of fingers on a hand). The details are discussed below.

The differences between the left and right plots in Figure .1 are caused by contributions from **`else`** and **`default`**. In the case of an if-if sequence the **`else`** can only appear on the final if-statement and a default-label be mixed with case-labels on the same statement sequence. The solid lines are a least-squares fit of the data to an exponential function.

2 Experimental setup

The experiment was run by your author during a 40 minute lunch time session at the 2009 ACCU conference (`www.accu.org`) held in Oxford, UK; between 275 and 325 professional developers attend this conference every year. Subjects were given a brief introduction to the experiment, during which they filled in background information about themselves, and then spent 27 minutes answering problems. All subjects volunteered their time and were anonymous.

The problem format was very similar in form to several previous ACCU experiments.[1]

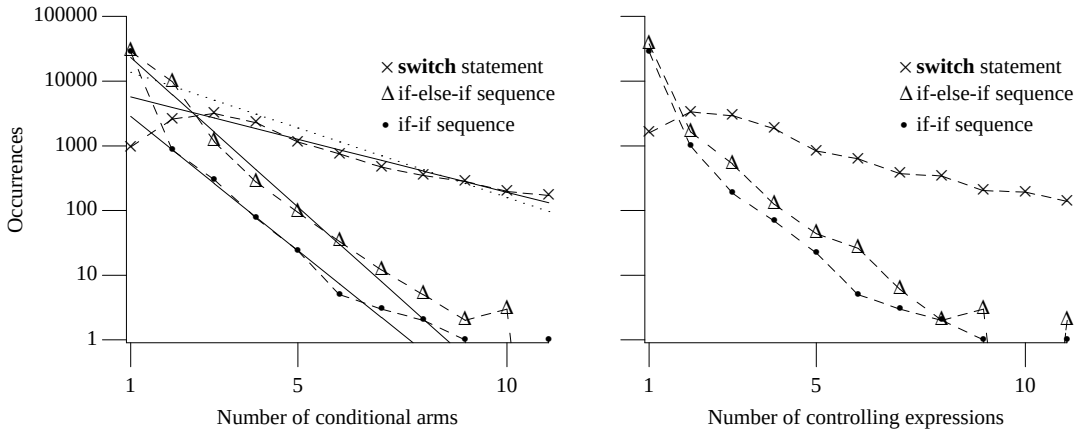

Figure .1: The left plot is of occurrences of if-else-if, if-if (uncertainty about which sequence a single if-statement, without an else-arm, belonged to was resolved by including it in both sequences) and switch-statements having the given number of conditional arms. The right plot is the number of controlling expressions in a if-else-if and if-if sequence and the number of case-labelled statement sequences (i.e., it excludes the effect of any **`default`**) in a switch-statement. The solid lines are a least-squares fit of the data to an exponential function. The dotted line is a fit to the sum of all sequences having a given number of arms.

2.1 The Problem

The following is an excerpt of the text instructions given to subjects (your author went through these instructions and the associated example once everybody had settled down in the room, prior to them answering any problem):

What you have to do

This is not a race and there are no prizes for providing answers to all questions. Please work at a rate you might go at while reading source code.

The task consists of remembering the sequence of three function calls and recalling this sequence later The function calls appear on one side of the sheet of paper and your response needs to be given on the other side of the same sheet of paper.

1. Read the function call sequence like you would when carefully reading lines of code in a function definition.
2. Turn the sheet of paper over. Please do **NOT** look at the function calls you have just read again, i.e., once a page has been turned it stays turned.
3. To create a time delay between reading the sequence of function calls and having to recall the sequence you are asked to write some code to assign a value to some variable. You do not know where these variables are defined, it may be in some other compilation unit, or locally within the current function.

   The function `set_variable` has one parameter and this can have any of the numeric values listed above the function skeleton, to the left. The value to be assigned to a particular variable appear to the right of the input value to which they apply.

   Only one function has to be written for each problem.

   The code can be written in a language of your choice (it would simplify subsequent analysis if either C, C++, Java, Pascal or C# were used).

   The values and variables either involve parts of the human body of relate to a person's home.
4. Once you have written the code you are now asked to recall the sequence of function calls seen on the previous page.

- if you remember the sequence circle the appropriate list,
- if you feel that, in a real life code comprehension situation, you would reread the original function call sequence, circle the *refer back* column on the right.

If you do complete all the questions do **NOT** go back and correct any of your previous answers.

```
1   op_1();
2
3   op_2();
4
5   op_3();
```

```
 1  ->  X = "Intel";
20  ->  y = "Motorola";
33  ->  W = "IBM";
41  ->  p = "Sun";
```

Two of the ways in which the appropriate assignment might be performed

```
1   void set_variable(int company)
2   {
3
4
5   if (company == 1)
6      X = "Intel";
7   else if (company == 20)
8      y = "Motorola";
9   else if (company == 33)
10     W = "IBM";
11  else if (company == 41)
12     p = "Sun";
13
14  }
```

```
void set_variable(int company)
{
 switch(company)
     {
     case 1: X = "Intel";
             break;
     case 20: y = "Motorola";
              break;
     case 33: W = "IBM";
              break;
     case 41: p = "Sun";
              break
     }
}
```

```
op_2();        op_3();        op_1();        op_2();
op_1();        op_1();        op_2();        op_3();        refer back
op_3();        op_2();        op_3();        op_1();
```

End of excerpt.

2.2 Background to problem generation

The problems and associated page layout were automatically generated using various `awk` scripts to generate troff, which in turn generated postscript. The source code of the scripts is available from the experiment's web page.[3]

2.2.1 Sequence of function calls

The names of the functions used for each sequence of three function calls was generated as follows:

- A total of 30 different sets of three function names was created. In seventeen of these sets the names were English words having the property that it was possible to place the names in an ordered relationship that many English speakers were thought likely to recognise, e.g., `toe`, `foot`, `leg`. Four of the sets contained English words having no obvious ordering relationship between them (e.g., `morning`, `collide`, `gutter`), five of the sets contained unrelated nonword-like sequences of letters (e.g., `cgy`, `pdl`, `kxr`) and four of the sets contained unrelated word-like sequences of letters (e.g., `esak`, `dard`, `sule`). The complete list of names used is available from the experiment's web page.[3]

- For each subject, the sequence of functional calls used in a problem was created by randomly selecting a previously unselected, set of three function names. The ordering of the names was randomised and this sequence was printed. The list of four possible answers was generated by creating three other unique random orderings of the names, each differing from the printed sequence, and printing out the four possible answers in a randomly selected order (*refer back* was added as a fifth possible answer).

2.2.2 Specification of function definition

The coding part of the problem answer sheets seen by subjects had two parts:

- a list of value/assignment pairs. When the function parameter had a given value a specified assignment was required to be executed.

  The problem description did not specify that any information on the kind of source statements to use.
- a function definition template within which enough white-space was provided for subjects to write their answer. The function took a single parameter whose name was intended to generate a particular semantic association in the subject's mind.

```
23  ->  K = 0 ;
46  ->  R = 1 ;
 4  ->  N = 2 ;
```

```
1  void set_variable(int num_ears_pierced)
2  {
3
4  // Sufficient vertical white space here to write code
5
6  }
```

It was decided that the number of conditions contained in each problem specification be either 3, 4 or 5. The detailed information present in Figure .1 was not available when this decision was made, otherwise problem specifications containing 2 conditions would also have been included.

Semantic information on the kind of operation being performed by the function was indicated via the name of the parameter variable being tested against and the numeric or string literal being assigned. For instance, the variable name `num_garden_sheds` is intended to convey to subjects that it holds a value representing a number of garden sheds (another possible interpretation is the number of garden sheds for sale or visible from some vantage point; both are open-ended in that the number of sheds is unlikely to be thought to be limited in real life to the range given in the problem), and requiring the assignment of one of the three strings `"two handed"`, `"one handed"` and `"hands free"` is intended to convey that the function involved humans using their hands (a closed set in the sense that humans have a maximum of two hands).

A total of 30 different possible variables and associated values were created. Half of these variables had names intended to indicate something involving the human body (the closed set) and the other half had names relating to human habitation (the open set). The ordering of the list of 30 possible variables was randomised for each subject.

The extent to which subjects considered it likely that the range of values held by a variable having a given name will change at a future date was found by asking them, at the end of the experiment: "For each of the problems you answered please specify what you think the likelihood is, on a scale of 1 to 10, that at some future date additional items will be added to the list of possible parameter values. 1 means extremely unlikely while 10 means inevitable." (the list of questions appeared in the same order as the list of problems they encountered).

The identifier used in each assignment statement was randomly chosen from a set of single letter identifier names and the order in which the assignment statements (for each problem) was listed was also randomized.

The complete list of names and corresponding literal values used, along with subjects' evaluation of the likelihood of additional values being added is available from the experiment's web page.[3]

3 Threats to validity

For the results of this experiment to have some applicability to actual developer performance (i.e., to be ecological valid) it is important that subjects work through problems at a rate similar to that which they would process source code in a work environment. Subjects were told that they are not in a race and that they should work at the rate at which they would normally process code. However, developers are often competitive and experience from previous experiments has shown that some subjects ignore the work rate instruction and attempt to answer all of the problems in the time available. To deter such nonwork-like behavior during this experiment the problem pack contained significantly more problems than subjects were likely to be able to answer in the available time (and this was pointed out to subjects during the introduction).

If subjects are asked to repeatedly write the same kind of coding construct situation they may not behave in the same way as when involved in having to use a variety of different constructs. In the ACCU experimental context it would not be practical to ask subjects to write lots of different kinds of code. Within the constraints of the ACCU experiment it was only practical to use two independent subproblems and it is hoped that this would be sufficient to prevent subjects rapidly falling into a nonwork-like fixed pattern of behavior.

The structure of the problem used follows a pattern that is often encountered when trying to comprehend source code: see information (and remember some of it), perform some other task and then perform a task that requires making use of the previously seen information. In this experiment the three activities were: remember some coding information, write some unrelated code and finally recognize the previously remembered information.

It is possible that when answering a series of problems having the same overall structure subjects may decide to use the same coding technique for each problem, making their answers unrepresentative of what they would have written outside of the context of this experiment.

The **`if/switch`** problem involved a component that relied on subjects making a semantic association with the name of a variable and making use of that information. An experiment that uses semantics as the control variable depends on subjects recognizing the appropriate semantic content in the problem being answered. If the anticipated semantic effect does not appear in the results one explanation is that subjects failed to extract the implied semantic information, another is that subjects extracted different information from that intended by the experimenter; a subject's failure to make use of the intended semantic information is also an explanation.

3.1 Subject strategies and motivations

Talking with subjects who have taken part in previous ACCU experiments uncovered that they had used a variety of strategies to remember information in remember/recall problems and had problem completion motivations that had not been anticipated. The analysis of the threats to validity of these experiments[1] discussed the question of whether subjects traded off cognitive effort on one subproblem in order to perform better on another subproblem, or carried out some other conscious combination of effort allocation between subproblems. To learn about strategies used during this experiment, after 'time' was called on problem answering, subjects were asked to list any strategies they had used (two sheets inside the back page of the handout had been formatted for this purpose).

4 Results

It was hoped that at least 30 people (on the day 12) would volunteer to take part in the experiment and it was estimated that each subject would be able to answer 20 problem sets (on the day 16.6 sd 4.1; a total of 199 answers) in 20-30 minutes (on the day 27 minutes).

The average amount of time taken to answer a complete problem was 97.6 seconds. No information is available on the amount of time invested in trying to remember information, answering the coding sub-problem, and then thinking about the answer to the sub-problem (i.e., the effort break down for individual components of the problem).

The average professional experience of the subjects was 12 years (standard deviation 7.4).

4.1 if/switch choice

All but one subject primarily always used either an if-else statement (2 subjects) or a switch-statement (9 subjects; a few subjects answered one question using an if-statement). One subject used an if-else statement when the specification contained three conditions and a switch-statement when the specification contained more than three conditions.

Based on the measurements used to plot Figure .1, with three conditional arms the probability of a switch-statement being used is 73% (the value with four conditional arms is 89%). The likelihood that 11 out of 12 subjects will primarily always use a switch-statement is very low.

The pattern of usage seen in the experiment answers would not generate the relative frequency of occurrences seen in the source code measurements. Possible reasons for this experimental behavior include:

- The relative frequencies seen in Figure .1 are caused by something other than developers basing their choice of **`if`**/**`switch`** statement usage on the number of conditional arms.
- In an attempt to improve their performance in the remember/recognise subproblem subjects decided to always give either an if-statement or a switch-statement answer (i.e., they assumed that by not making a separate choice for each answer they were more likely to correctly answer the function sequence subproblem).

After time was called on answering problems subjects were asked to: “Please list any strategies you used when writing the code”

Strategies listed included: ’Always the same’, ’Dumb way, because little info (followed example)’, ’do not analyse context’ and ’Minimise amount of writing’. The subject who used a mixed **`if`**/**`switch`** strategy wrote: ’3 items or less -> if-else chain, more items -> switch case’.

Subjects used two orders for testing the parameter against the various numeric values listed in the problem specification:

- The two subjects who primarily used if-else tested the numeric literals in the order in which they appeared in the specification. The subject who only used if-else in the three conditional arm problems sorted the numeric values and tested them in this order.
- Seven of the nine subjects who primarily used **`switch`** tested the numeric literals in the order in which they appeared in the specification, the other two subjects plus the subject who used an if-else in the three conditional arm problems sorted the numeric values and tested them in this order.

4.2 Recognition performance

Subjects saw a sequence of three function calls and later had to either select one out of four sequences or select *refer back*. One of the four sequences was the same as that seen earlier, so there a random answer had a 25% chance of being correct.

Of the 199 problems answered by the 12 subjects 84.9% were correct, 12.6% were *refer back* and 2.5% incorrect (in all 5 incorrect cases the first function name in the answer sequence was correct).

All of the incorrect answers were given by just 25% of the subjects; this is not statistically significant because of the very small number of incorrect answers involved.

The *refer back* answer was not given by 58% of subjects, but was given quiet a few times (mean 31.5% of answers) by 33% of the subjects. There are several possible explanations for some subjects giving many *refer back* answers, including:

- Self-knowledge, or metacognition, is something that enables a person to evaluate the accuracy of the memories they have. Perhaps these subjects have poor metacognitive abilities (i.e., they underestimated the accuracy of their memories and might have been correct had they risked giving an answer),
- these subjects were very cautious, or risk averse, people,

- these subjects exhibited poor short term memory performance during the experiment (which may be due to being tired or having a low short term memory capacity).

After time was called on answering problems subjects were asked to: "Please list any strategies you used to remember the sequence of items"

Strategies commonly listed by subjects for remembering a sequence of function definitions included: noting if the items were in alphabetic order, reverse alphabetic order, logical order and noticing when the initials formed an acronym they knew. These strategies implied that subjects were not remembering the names of the functions but some pattern that could be used to later recognise which sequence to circle (for their answer). This method of operation works because subjects were only asked to remember one sequence at a time, something that is not that common in a software development environment.

5 Conclusion

The results did not produce any evidence that significantly supported the hypothesis concerning developer choice of **`if`**/**`switch`** statements or that a known ordering relation between a sequence of functions names aided later recognition of that sequence.

Subject made so few mistakes in the function sequence recognition problem that it is not possible to reliably detect any patterns in the mistakes made. It is difficult to know how to structure a recognition problem that would generate a sufficient number of subject mistakes while maintaining a reasonable degree of ecological validity.

The memory problem could have been structured as a recall problem (i.e., ask subjects to write down the sequence of names without being given any external clues; when writing software developers have to recall the appropriate sequence to call functions and when reading existing source spot when a sequence of calls is incorrect). This experiment was not based on recall because it was thought, prior to the experiment, that this would require too much cognitive effort from subjects who might then give many *refer back* answers.

All but one subject used the same coding construct for all **`if`**/**`switch`** problem answers. This behavior may have been an artifact of the experimental situation. Future experiments investigating **`if`**/**`switch`** decision making might like to include problems containing two conditional arms. When describing the problem to subjects and telling them what they are being asked to do it might be worthwhile stressing that they should give the **`if`**/**`switch`** part of the experiment equal weighting and not use a coding strategy aimed at improving their performance on other parts of the problem.

6 Further reading

For a readable introduction to human memory see "Essentials of Human Memory" by Alan D. Baddeley; a more advanced introduction is given in "Learning and Memory" by John R. Anderson. An undergraduate level discussion of some of the techniques people use to solve everyday problems is provided by "Simple Heuristics That Make Us Smart" by Gerd Gigerenzer and Peter M. Todd. An excellent introduction to many of the cognitive issues that software developers encounter is given in "Thinking, Problem Solving, Cognition" by Richard E. Mayer.

6.1 Acknowledgements

The authors wishes to thank everybody who volunteered their time to take part in the experiments and ACCU for making a slot available, in which to run the experiment, at the 2009 conference.

References

1. D. M. Jones. Developer beliefs about binary operator precedence. *C Vu*, 18(4):14–21, Aug. 2006.
2. D. M. Jones. Deciding between **if** and **switch** when writing code. *C Vu*, 21(5):14–20, Nov. 2009.
3. D. M. Jones. Experimental data and scripts for deciding between if and switch. http://www.knosof.co.uk/cbook/accu09.html, 2009.