Developer categorization of data structure fields (part 1 of 2)

Experiment performed at the 2008 ACCU Conference
First published in CVu vol. 20 no. 6

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

1 Introduction

Data structures are an important aspect of software engineering and while a great deal of effort has been expended on analysing them from a mathematical point of view almost no effort has put into analysing the thought processes used by developers when creating data structures.

This article investigates the decision making process behind developers' creation of datatype definitions. Two sources of data are analysed, measurements of existing code (in particular patterns of field ordering within one or more aggregate types, e.g., C **`struct`** types) and the results of an experiment carried out at the 2005 and 2008 ACCU conferences (this asked subjects to create one or more data structures to handle a specified collection of data items).

This first part investigates various patterns that occur in the definitions contained in a large body of existing C source code, while a second part discusses the results of an experiment that asked developers to create a set of definition from a specification.

An understanding of the thought processes used by developers when writing code is essential to the creation of support tools (e.g., refactoring or flagging suspicious usage) and coding guidelines.

From the human point of view the organization of data structures is a classification problem. People actively use classification to infer characteristics of objects they have not encountered before. For instance, if I encounter a four legged animal that barks and has a tail I am likely to classify it as a dog and I can use my existing knowledge of objects that I have previously placed in the dog category to infer some of the likely characteristics of the animal I have just encountered. This process is not perfect, for instance during World War II some of the children evacuated from the built-up areas of London to the country had not seen sheep before and initially classified them as dogs.

Children as young as four have been found to use categorization to direct the inferences they make,[3] and many different studies have shown that people have an innate desire to create and use categories (they have also been found to be sensitive to the costs and benefits of using categories[9]).

Analysing the factors that influence how developers organise information into one or more aggregate datatypes requires a more global perspective than can be obtained from measuring source code (e.g., background data on the application domain and program design constraints).

This first part of the article analyses existing C source code, concentrating on finding common usage patterns within individual definitions. Experience with source code suggests that the following are some of the patterns that occur in the ordering of fields within aggregate definitions:

- Fields having the same type will be placed next to other fields having the same type.
- Fields are likely to be placed close to other fields containing information with which they share semantic associations.
- The ordering of fields will follow the order in which information appears in any written specification used as the basis for creating a structure definition.

This second part of the article uses the experimental results to analyse relationships between information that the designer perceives in the application (e.g., in a geospatial application it is likely that the various kinds of location information will be a strong candidate for being grouped together) and various kinds of housekeeping information internal to the program (e.g., the `next` field in a linked list) will also drive the content and form of these data structures.

The experiment performed at the two conferences asked subjects to create data structure to represent various kinds of information. The results of these two experiments are discussed in the second part of this article.

2 Analysis of existing source

The following subsection discusses measurements of **`struct`** definition field usage in the translated form of a number of large C programs (e.g., gcc, idsoftware, linux, netscape, openafs, openMotif, and postgresql).

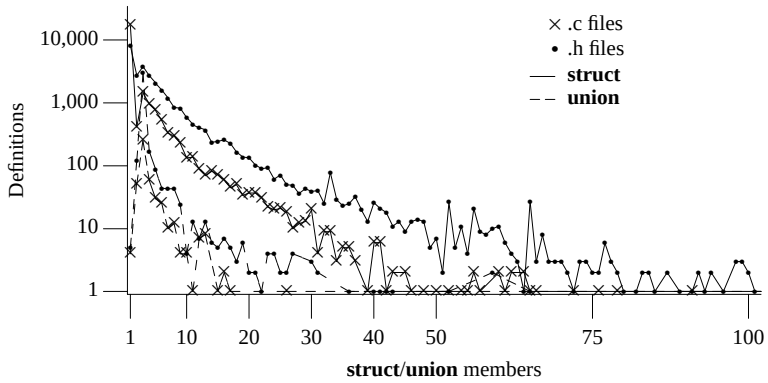

Figure .1: Number of structure and union type definitions containing a given number of members (members in any nested definitions are not included in the count of members of the outer definition). Based on the visible form of the `.c` and `.h` files.

The contents of any header files were only counted once. This C source yielded 6,244 **`struct`** definitions containing a total of 47,554 fields (average 6.7 per definition).
The following patterns of behavior were investigated:

- Field type sequences, in particular the observation that fields having the same type are often placed next to each other.
- Shared field names. The extent to which an aggregate definition contains one or more fields whose names also appears in other aggregate definitions.
- Character sequences that are shared between fields in the same aggregate definition, e.g., `farm_house` and `farm_animal` both contain `farm`.
- Influence of specification on field ordering. When a written specification exists do developers follow the ordering in which it presents information when ordering fields within a structure definition?

2.1 Field type sequences

Experience shows that fields having the same type are often placed next to each other within a structure. This subsection investigates this observation.
If, within a aggregate definition, fields of the same type are always placed next to each other, then a definition containing T different field types would have $T - 1$ changes of type. If developers always attempt to place fields of different type next to each other the number of changes of type has a possible maximum value equal to the number of fields present minus one (this pattern requires sufficient fields of different type that it is possible to continually change type).
The possible sequences of field type within a definition will depend on the number of occurrences of each type. For instance, a definition containing four fields with two fields sharing one type and the other two fields sharing a different type (say x and y) can have one of six possible type sequences xxyy xyxy yxxy yxyx xyyx yyxx of which 33.3% have the minimum number of changes of type. When two fields share a single type and each of the other two fields have different types there are 12 possible field type sequences xxyz xxzy xyxz xyzx xzxy xzyx yxxz yxzx yzxx zxyx zxxy zyxx and again 50% of the sequences have the minimum number of changes of type.
As the number of fields increases the number of possible type sequences increases. In the case of five fields with three sharing the same type and two sharing a different type there are 10 possible type sequences xxxyy xxyxy xxyyx xyxxy xyxyx xyyxx yxxxy yxxyx yxyxx yyxxx of which 20% of the sequences have the minimum number of changes of type.

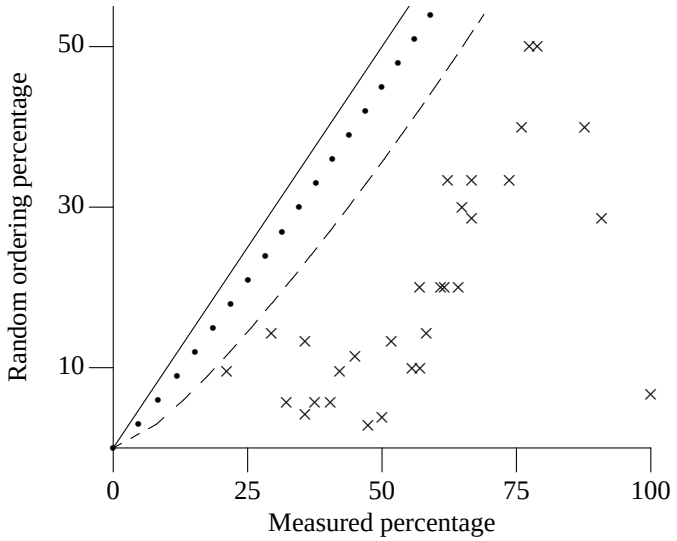

Figure .2: The measured percentage of **`struct`** definitions having a minimal type change sequence (x-axis) and the percentage that expected to occur if fields were ordered randomly. A cross indicates a measured percentage and the random percentage for a definition containing that particular number of types. If field ordering were random the crosses would be expected to cluster along the solid diagonal line (bullets indicate 1 standard deviation, dashed line 3 standard deviations). Data for all definitions containing between four and seven (inclusive) fields.

Table -1.1 lists possible type sequences for aggregates containing 4, 5 and 6 fields, their number of occurrence in C **`struct`** definitions, the expected percentage having the minimal number of changes of type and the actual percentage having minimal changes of type. The enumeration of field type sequences was calculated using a purpose written program.[1]

All pointer types were treated as being the same type (i.e., the pointer type), all arrays were treated as having the same type (i.e., the array type) and all struct types were treated as having the same type (i.e., the struct type; it did not matter whether these types occurred via the use of a typedef on an inline definition).

The number of **`struct`** definitions extracted from the sourced used for this analysis was sufficient to provide enough data to analyse definitions containing seven or fewer fields. Figure -1.1 shows a rapid, power-law like, decline in the number of C **`struct`** definitions as the number of fields in a definition increases, so significantly more source code would be needed to analyse definitions containing eight or more fields.

In all cases the number of definitions containing minimal field type change sequences was greater than would have occurred had the selection been random, Figure -1.2 shows the percentage is greater than three standard deviations (calculated as $sqrt(p(1-p)n)$ where p is the probability of a minimal field sequence occurring and n is the number of instances, which has been normalised to 100).

Future research might analyse definitions where the number of type changes was only one, or more, greater than the minimum.

[1] This program was written by Dawn Lawrie. Your author is keen to hear from anybody who knows of an algorithm, other than enumeration, for calculating these values. The number of different type combinations for a definition containing N fields is the number of integer partitions of N. The probability of encountering a field order having the minimum number of changes of type is obtained by dividing the number of minimum changes of type (this is $m!$, where m is the number of types in the partition) by the number of possible combinations of each type in the partition, for each of partition possible for a given definition.

Table .1: Field type patterns for **struct** definitions containing various numbers of fields, the number of measured occurrences oin source code and the measured percentage having a minimum number of type changes, and the percentage of type changes expected if field ordering was random. When all fields have the same type, or where each field has a different type, every sequence contains the same number of type changes and were not counted.

Fields	Field type pattern	Source occurrences	% minimum type change	% random ordering
4	1 1 2	239	77.4	50.0
4	1 3	185	78.9	50.0
4	2 2	98	62.2	33.3
5	1 1 1 2	57	87.7	40.0
5	1 1 3	94	64.9	30.0
5	1 2 2	86	57.0	20.0
5	1 4	121	76.0	40.0
5	2 3	94	61.7	20.0
6	1 1 1 1 2	19	73.7	33.3
6	1 1 1 3	23	60.9	20.0
6	1 1 2 2	28	35.7	13.3
6	1 1 4	53	64.2	20.0
6	1 2 3	72	55.6	10.0
6	1 5	51	66.7	33.3
6	2 2 2	9	100.0	6.7
6	2 4	60	51.7	13.3
6	3 3	21	57.1	10.0

Possible reasons for this grouping of fields by type include:

- Fields are grouped together semantically and such related fields are likely to hold similar kinds of information and this is represented using the same type.

- Developers are attempting to minimise unused storage space. Some processors require that types larger than a byte be assigned an address that is a multiple of some power of two. For instance, an **int** may have to be assigned an address that is a multiple of 2 or 4. Grouping fields having the same type is a simple strategy to ensure that no unused padding appears between these fields.

- Developers treat the type of a field as a categorization attribute and give it significant weight when deciding the relative ordering of fields within a structure type.

2.2 Same information, same name and type?

Developers are encouraged to give meaningful names to identifiers. If fields in different definitions have the same name (and contain a reasonable number of characters) it might be supposed that the information they hold is closely related semantically and these fields might be expected to share the same type.

This supposition relies on different developers representing the same information using the same name and the same type (the source code used for this analysis was written by a large number of developers). An earlier ACCU experiment[7] that asked subjects to assign a meaning to different identifier names found a wide range of meanings assigned to each identifier. The analysis of the experimental results, in part two of this article, will provide information on the extent to which developers choose the same name and type to denote the same application information.

Measurements of the translated C source show that within the measured 6,244 **struct** definitions and 47,554 fields, 21,805 (45.9%) fields had names that were not shared by other fields, while 6,415 names were shared by two or more of the other 25,749 fields (average 4.0 fields per name).

Table .2: Number of times a given name occurs as a field and the number of different names occurring that number of times (e.g., if `blah_blah` is used three times as the name of a field it would add one to the count appearing in the second column of the 3's row). Only field names containing 4 or more characters were analysed.

Times name used	Number of different names	Percentage of all shared names
2	3239	50.54
3	1108	17.29
4	622	9.71
5	310	4.84
6	290	4.52
7	162	2.53

Is a field that shares its name with another field more likely to have a certain type? With one possible exception Table -1.3 suggests that the type of a field does not change the probability of its name appearing in another **`struct`** definition. A possible exception is pointer types which appear to be slightly more likely to occur, than the average, as the type of a field whose name appears in another **`struct`** definition (your author is uncertain how to appropriately calculate a standard deviation for values appearing in the two columns and so cannot say anything about statistical significance).

Table .3: Occurrences of structure type changes as a percentage of all field types (second column) and as a percentage of all fields whose names also appear in other **`struct`** definitions.

Type	% all occurrences	% duplicate occurrences
int	16.1	16.8
pointer-to	14.1	18.3
unsigned char	11.6	11.1
unsigned int	10.3	8.7
array of	9.4	10.9
unsigned short	7.6	7.5
struct	7.2	7.3

A study by Anquetil and Lethbridge[1] analyzed 2 million lines of Pascal (see Table -1.4) and found that members that shared the same name were found to be much more likely to share the same type than members having different names.

Table .4: Number of matches found when comparing between pairs of members contained in different Pascal records. Adapted from Anquetil and Lethbridge.[1]

	Member Types the Same	Member Types Different	Total
Member names the same	7,709 (33.7%)	15,174 (66.3%)	22,883
Member names different	158,828 (0.2%)	66,652,062 (99.8%)	66,710,890

Refactoring and the desire for backwards compatibility can result in fields with the same name having the same type, i.e., fields moved into a new definition are retained in the original definition for backwards compatibility (perhaps because it is contained in a header included by lots of third party programs).

2.3 Fields sharing a subsequence of characters

Developers sometimes use several words or abbreviations to create an identifier having what they consider to be an appropriate semantic interpretation. If two or more fields within an aggregate definition share a semantic association it is possible that one or more subcomponents of their names will share a sequence of characters, e.g., `farm_house` and `farm_animal` both contain `farm`. The article on the 2007 ACCU experiment[8] describes an algorithm for extracting the likely intended subcomponents of an identifier and this was used for the analysis described here.

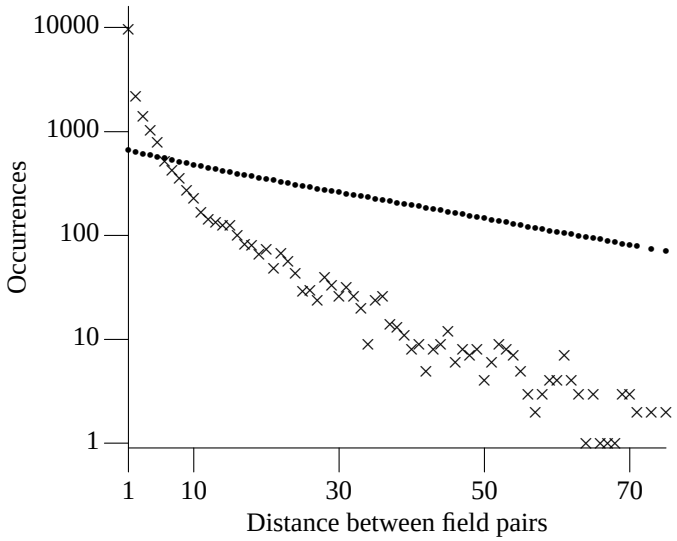

Figure .3: The number of individual field pairs that share one or more subcomponents plotted against their relative distance from each other; based on the translated C source (crosses) and based on random ordering of field pairs (bullets). Only definitions containing five or more fields were analysed and only character sequences containing three or more characters were counted. A field pair was only counted once, i.e., field pairs sharing more than one subcomponent were not counted multiple times.

A pair of fields within the same aggregate definition that share one or more is subcomponents is called a *field pair* in this article.

If developers group together semantically related fields within a definition then field pairs will be closer to each other than if fields were ordered randomly. This grouping expectation relies on there being a sufficiently large number of different subcomponent character sequences that developers are not likely to assign different semantic interpretations to the same sequence.

Figure -1.3 shows the number of individual field pairs (crosses) that share one or more subcomponents plotted against their distance from each other (adjacent fields have distance 1, if one field separates them they have distance 2 and so on; see Table -1.5). The bullets show the behavior expected if the fields in a field pair are ordered randomly.

Figure -1.4 shows the average distance of all field pairs (crosses) occurring within a definition having a given number of fields. The bullets show the average distance expected if the fields in a field pairs are ordered randomly.

The average distance slowly increases as the number of fields in a definition increases. Reasons for this increase, rather than staying essentially the same, include:

- Definitions containing lots of fields are more likely to contain accidental field pairs, in the sense that they share a subsequence that was not explicitly intended to be shared, than definitions containing few fields.
- The greater the number of fields the greater the probability that many semantically related fields will be clustered together, creating field pairs over a greater distance.
- Fields are added and deleted from definitions as software evolves.[11] In some cases developers have to add new fields at the end of a definition so as not to change the storage layout of the existing fields. In this case a newly added field may create a field pair but not be placed much further away from its corresponding member(s) than would be the case if it had been added when the definition was first created.

If field pairs are clustered together it is to be expected that the average field pair distance will be less than

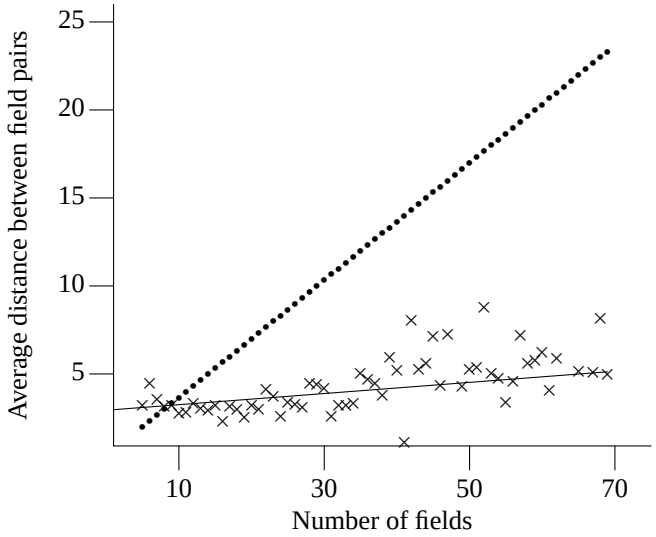

Figure .4: The average distance between field pairs for definitions containing a given number of fields; based on the translated C source (crosses) and based on random ordering of field pairs (bullets). The straight line is a least squares fit of the average distance measurements. Only definitions containing five or more fields were analysed and only character sequences containing three or more characters were counted. A field pair was only counted once, i.e., field pairs sharing more than one subcomponent were not counted multiple times.

the random field distance for all sizes of definitions. Figure -1.4 shows that this is not the case when the definition contains between five and eight fields: 5 fields, expect less than 2.0 but find 3.25; 6 fields 2.33 vs. 4.46; 7 fields 2.67 vs. 3.54; 8 fields 3.00 vs 3.18. Possible reason for this behavior include: for smaller definitions developers feel the semantic association between fields more obvious than larger definitions and there is less need to explicitly call it out via shared subcomponent names; in a set of fields having a tight semantic association any shared name is likely to be part of the aggregate type name and repeating this sequence in field names would be overkill.

2.3.1 Average distance between random pairs of fields

The average distance between a randomly chosen pair of fields in a definition containing N fields is $(N+1)/3$.

Table .5: Number of possible occurrences (column values) of each *distance* (top row) between every possible combination of field pairs in definitions containing a given number of fields (left column). For instance, a **`struct`** containing five fields has four field pairs having distance 1 from each other, three distance 2, two distance 2, and one field pair having distance 4.

Distance	1	2	3	4	5	6
Number of fields						
4	3	2	1			
5	4	3	2	1		
6	5	4	3	2	1	
7	6	5	4	3	2	1

Proof: The average distance can be obtained by summing the distances between all possible field pairs and dividing this value by the number of possible different pairs.
Table -1.5 shows the pattern that occurs as the number of fields in a definition increases.
In the case of a definition containing five fields the sum of the distances of all field pairs is: (4*1 + 3*2 + 2*3 + 1*4) and the number of different pairs is: (4+3+2+1). Dividing these two values gives the average distance between two randomly chosen fields, e.g., 2.
Summing the distance over every field pair for a definition containing 3, 4, 5, 6, 7, 8, ... fields gives the

sequence: 1, 4, 10, 20, 35, 56, ... This is sequence A000292 in the On-Line Encyclopedia of Integer Sequences, www.research.att.com/{~}njas/sequences and is given by the formula $n*(n+1)*(n+2)/6$ (where $n = N - 1$, i.e., the number of fields minus 1).
Summing the number of different field pairs for definitions containing increasing numbers of fields gives the sequence: 1, 3, 6, 10, 15, 21, 28, ... This is sequence A000217 and is given by the formula $n*(n+1)/2$.
Dividing these two formula and simplifying yields $(N+1)/3$.

3 Influence of a specification

Many software development projects start with some form of specification containing information about the application. This specification may be sufficiently detailed that it is possible to create a one-to-one mapping from the information it contains to an aggregate definition. Of necessity any information present in a specification, that may be encoded in the fields of a definition, appears in some order. Definitions are created by adding one field at a time and if the developer works systematically through a specification it is to be expected that field ordering used will have that order. Following the order of information in a specification has the following advantages:

- simplifies the process of verifying that all items listed in the specification are covered by a field,
- requires less cognitive effort to make use of an existing ordering than to create a different one,
- people influenced by what appears to be an existing way of doing things.

To what extent does the order of fields in **`struct`** definitions in existing code follow the order of corresponding information within a specifications. Source of data that can be used to answer this question are hard to come by. National and international standards sometimes go into the level of detail required and in some cases there are implementations available whose source code can be viewed. For instance, POSIX[5][6] provides a specification of the information that a conforming implementation is required to support in various **`struct`** definitions. POSIX is supported by a wide variety of vendors who need to make available to their customers the source code, in header files, of the specified definitions.[2]
There are a number of complicating factors involved in using POSIX for this analysis, including:

- Implementations of much of the functionality specified in POSIX were available before work started on this standard. The authors of the standard may have based some of the orderings given in the specification on one or more of these existing implementations.
- One of the original vendors (i.e., AT&T) licensed various versions of its implementation to other vendors. It is possible that this resulted in some of the contents of some headers sharing more in common than they might have if implemented from scratch.

Header files from five different platforms supporting the functionality specified by POSIX were analysed (value in parentheses is the latest year any of the headers, for that implementation, could have been modified). The platforms were: RedHat 5.0 Linux (1996), Solaris 2.5 (1996), HP/UX 10 (1996), RS/6000 running AIX 3.2 (1992), Suse 10.3 Linux (2008).
Two versions of the POSIX Standard were used for the analyse, the first[5] published in 1990 and the latest[6] published in October 2008 (current a final draft being voted as the next revision of POSIX).
Most of the **`struct`**s are required to support a relative small number of fields (the number in parentheses): `sigaction` (3), `uname` (5), `tms` (4), `stat` (10), `flock` (5), `termios` (5) and `passwd` (5).

- With the exception of Solaris, which had one field out of order, all implementations of the `sigaction`, `uname` and `tms` structure types had the same field ordering as that given in the POSIX standard.

[2] Your author is keen to hear from anybody who knows of any other publicly available specifications and associated implementation headers.

- In all implementations the `flock` (one additional field in Solaris, two additional fields in AIX) and `termios` (the same three additional fields in RedHat and Suse) structure types had the same field ordering as that given in the POSIX standard.
- Within the `stat` structure type defined in the header `sys/stat.h` the relative positions of the fields `st_mode` and `st_dev` was reversed between the first and latest edition of the POSIX Standard. Some vendors follow one ordering while the remaining vendors follow the other. All vendors have defined more than the ten fields listed in the first edition and intersperse the additional fields at a variety of different locations.
- Within the `passwd` structure type defined in the header `pwd.h` all of the implementations included fields not specified in either version of the POSIX standard, with the additional field shared across implementations always occurring in the same relative location.
- The `sigevent` structure type was not specified in the first edition of the POSIX standard, but is in later versions. It is defined in Solaris and HP/UX (in both cases with one field missing) and Suse (with two fields appearing in a different order).

4 Other definitions

It is likely that aggregate definitions will not be the only kind of definitions created by subjects taking part in the experiment. Names may be given to scalar types through the use of **`typedef`** (or some other mechanism appropriate to the language used by the subject) and names may be defined to represent particular kinds of entities (e.g., using macros or enumeration constants).

These non-structure definitions are not of interest to the questions investigated in this study, but they may be of interest for other questions concerning developer decision making. For instance, whether to use a macro (e.g., a `#define`; one study[4] was able to automatically map some sequences of identifiers defined as object-like macros to members of the same, tool created, enumerated type).

5 Discussion

The first part of this study set out to investigate the extent to which various patterns of **`struct`** field usage occur within existing source code.

- Fields that have the same type were found to occur in sequence with a probability that is significantly better than chance. There is no data to support any conclusions for why this might be so (e.g., developers actively try to achieve such a clustering or that clustering is driven by developer perceived semantic similarity between the information contained in fields and semantically similar information tends to have the same type).
- Fields in different definitions are sometimes given the same name (see Figure -1.2). There is no data to support any conclusions (e.g., developers use the same name to represent semantically similar information or the probability that different developers happen to use the same names). Fields having a particular type were not more likely to share the same name (see Figure -1.3). Refactoring and a desire for backwards compatibility can generate fields of the same name having the same type.
- Pairs of fields, in the same definition, whose names share a common subcomponent are much closer to each other than would be expected from a random field ordering. One explanation is that fields sharing a semantic association are grouped together and share one or more sequence of characters in their name.
- In a very small sample there was very good agreement between the ordering of information in a specification and the fields in structure definitions used by various implementations.

5.1 Threats to validity

The C source measured for this study has been actively worked on for many years and during that time its **`struct`** definitions are likely to have evolved. A study[11] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 79% of releases one or more existing structure or union types had one or more fields added to them, while structure or union types had one or more fields deleted in 51% of releases and had one or more of their field names changed in 37% of releases. One or more existing fields had their types changed in 35% of releases.[10]

The only source code measured was written in C. To what extent is it possible to claim that the findings apply to code written in other languages? Measurements of classes[12] in large Java programs have found that the number of members follows the same pattern as that in measured in C (see Figure -1.1). While there are no obvious reasons why the patterns found in C should not also occur in other languages, there is no reason why they should. Measurements of source written in other languages would put this issue to rest.

6 Further reading

A readable collection of papers on how people make use categories to solve problems quickly without a lot of effort: "Simple Heuristics That Make Us Smart" by Gerd Gigerenzer, Peter M. Todd and The ABC Research Group, published by Oxford University Press, ISBN 0-19-154381-7.

A readable upper graduate level book dealing with how people create and use categories "Classification and Cognition" by W. K. Estes, published by Oxford University Press, ISBN 0-19-510974-0.

7 Acknowledgements

The author 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 both conferences.

Thanks to Dawn Lawrie and David Binkley for commenting on an earlier draft and Dawn for writing a program to evaluate field type sequences.

References

1. N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of CASCON'98*, pages 213–222, 1998.
2. M. Celce-Murcia and D. Larsen-Freeman. *The Grammar Book: An ESL/EFL Teacher's Course*. Heinle & Heinle, second edition, 1999.
3. S. A. Gelman and E. M. Markman. Categories and induction in young children. *Cognition*, 23:183–209, 1986.
4. J. M. Gravley and A. Lakhotia. Identifying enumeration types modeled with symbolic constants. In L. Wills, I. Baxter, and E. Chikofsky, editors, *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 227–238. IEEE Computer Society Press, Nov. 1996.
5. ISO. *ISO/IEC 9945-1:1990 Information technology —Portable Operating System Interface (POSIX)*. ISO, 1990.
6. ISO. *ISO/IEC FDIS 9945:2008 Information technology —Portable Operating System Interface (POSIX®)*. ISO, 2008.
7. D. M. Jones. I_mean_something_to_somebody. *C Vu*, 15(6):17–19, Dec. 2003.
8. D. M. Jones. Experimental data and scripts for operand names influence operator precedence decisions. `http://www.knosof.co.uk/cbook/accu07.html`, 2008.
9. W. T. Maddox and C. J. Bohil. Costs and benefits in perceptual categorization. *Memory & Cognition*, 28:597–615, 2000.
10. I. Neamtiu. Detailed break-down of general data provided in paper[11] kindly supplied by first author. Jan. 2008.
11. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.
12. R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 45–54, Sept. 2003.