Information Clustering for Data Structure Definitions

David Binkley
Loyola University
Baltimore MD
21219, USA
binkley@cs.loyola.edu

Derek M. Jones
Kingston University
Kingston-upon-Thames,
Surrey
KT1 2EE, UK
derek@knosof.co.uk

Dawn J. Lawrie
Loyola University
Baltimore MD
21219, USA
lawrie@cs.loyola.edu

ABSTRACT

Consistency is important to the success of any software project and this includes the organization of its data structure definitions. Thus there is a need to better understand the decisions programmers make when clustering items into data structures. Results from an experiment that asked programmers to design data structures based on various specifications are presented. The results show that although there is significant variation among subjects, patterns exist across designs. Participants included 26 professional and 12 student programmers. Comparisons of the two groups indicate that students are more conformist to the specification, but also show greater diversity in their answers, when compared to professionals. These results help to characterize the learning that experience brings and suggest that when selecting programmers for a task involving data structure design or modification, each programmer's *layout style* be taken into account.

1. INTRODUCTION

This paper describes an experiment that is part of research aimed at understanding and exploiting connections between the characteristics of human mental processing obtained in cognitive psychology and software engineering. The experiment investigates the decisions made by software developers when asked to organize items of information into data structures (e.g., structure fields within a C struct).

Experienced software developers sometimes criticize the way in which novice developers organize data structures. Is this criticism simply a case of personal preference or does experience teach developers something that causes them to organize structures differently than novices? To address this question, the experiment contained two groups: experienced and novice programmers.

For even a small number of data items there are a huge number of possible combinations. For instance, there are 3^{10} ways in which 10 items can be split between three structure definitions. The two primary aims of the experiment include understanding the extent to which programmers make similar decisions when designing data structures and investigating the impact that the specification has on the ordering of items within data structures.

Consider the factors that influence the effectiveness of an API to be used by many programmers. For example, how easy is it to remember what information is held in which structure definition, does one structure contain all of the related information (e.g., does a function need a single parameter having that type or multiple parameters)? A second factor is the consistency between the developers who originally created a structure and those tasked with its modification. If these two groups would make radically different organizational decisions, then the cohesiveness (as judged by the consistency of information clustering) will be reduced.

The experiment presented in this paper considers these questions. It thus provides a baseline for understanding the choices made and variety of data structure layouts created by programmers. Furthermore, the research suggests several future studies. For example, evaluating whether developers use less cognitive effort maintaining software written by those having similar organizational preferences. This kind of attention to data structure organization during both initial development and subsequent maintenance should make information organization more internally consistent and therefore require less effort from subsequent developers.

The remainder of the paper first presents some necessary background information in Section 2 before describing the experiment's design including the four hypotheses considered, and the results in Sections 3 and 4, respectively. This is followed by a discussion of related work in Section 5 and threats to validity in Section 6. Finally, Section 7 summarizes the paper and suggests future work.

2. BACKGROUND

This section first introduces some basic terminology used in the paper. It then looks at how humans organize related information – first in general and then in the specific case of data structures. In this paper the term *specification* is used to refer to the description of the items to be included in a program's data structures. A specification consists of a collection of information *items* (e.g., "Size of garden"). In building data structures, a programmer represents a specification as a collection of *data structure definitions* (e.g., C structs, or Java or C++ classes). A data structure definition is built from a collection of *fields* (e.g., "garden_size"). Each field denotes the name and type the designer assigns to an item from the specification.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HAoSE2009 '09 Orlando, Florida USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

The general problem of how people go about the process of clustering items into categories (which is one way to think about how fields are organized into data structures) has been extensively studied by psychologists [12]. A brief overview of some of the key findings from these studies is given. By dividing objects into categories, people reduce the amount of information they need to learn by effectively building representations that enable them to lookup information on an object based on its category membership. This approach also helps them deal with objects they have not encountered before.

What method do people use to decide which, if any, category an object is a member of? The following are four popular theories [7, 10]. Each is first introduced in general terms and then followed by a brief description of its potential impact on data structure formation.

- The *exemplar-based theory* of classification proposes that specific instances, or *exemplars*, act as prototypes against which other members are compared. Objects are grouped, relative to one another, based on some similarity metric. With the exemplar-based theory, specific instances are the norm against which membership is decided. When asked to name particular members of a category, the attributes of the exemplars are used as cues to retrieve other objects having similar attributes. In data structure design, use-cases generate a bias towards this kind of organization as they tend to focus designers on a few examples.

- The *defining-attribute theory* proposes that members of a category are characterized by a set of defining attributes. This theory predicts that these attributes should divide objects up into different concepts whose boundaries are well defined. All members of the concept are equally representative. Object-oriented designs that adhere strongly to the strict tenants of design level objects identifying with objects in the real-world, are a good match with this theory.

- The *explanation-based theory* of classification proposes that there is an explanation for why categories have the members they do. For instance, the biblical classification of food into *clean* and *unclean* is roughly explained by saying that there should be a correlation between type of habitat, biological structure, and form of locomotion: creatures of the sea should have fins, scales, and swim (sharks and eels don't) and creatures of the land should have four legs (ostriches don't). From a predictive point of view, explanation-based categories suffer from the problem that they may heavily depend on the knowledge and beliefs of the person who formed the category; for instance, the set of objects a person would remove from their home while it was on fire. A system that has had significant upfront requirements and analysis work provides designers with the necessary knowledge and background to produce a logical organization in its data structures and thus is a good match with this theory.

- Finally, the *prototype theory* proposes that categories have a central description, the prototype, that represents the set of attributes of the category. This set of attributes need not be necessary or sufficient to determine category membership. The members of a category can be arranged in a typicality gradient, representing the degree to which they represent a typical member of that category. The prototype theory matches the process used in producing a generalization-specialization hierarchy for use in an object-oriented program.

Finally, this section considers factors that influence data-structure design. When a programmer is presented with a specification he or she typically relies on his or her experience to design data structures to hold the required information. Doing so involves the need to estimate the trade-offs among a large number of possible structure definitions. The following are some of the factors that influence these trade-offs:

- *Simplicity.* Writing code is simplified when all required information can be obtained by accessing a single structure.

- *Minimizing the amount of storage used during program execution.* For instance, consider a tree whose nodes include partially shared information. Replacing a single tree-node structure with two structures allowing references to the shared information to replace copies, can reduce a program's memory footprint [3].

- *Information hiding.* It may be necessary to restrict access to some information, perhaps for commercial or security reasons, or to reduce the likelihood that a fault in one part of a program can effect unrelated data in another part.

- *Minimizing the impact of future updates.* Good data structure design makes it easier to enhance or fix a system. For example, when organized into separate structures based on semantic relatedness, adding a new field or changing an existing field, means that only code referencing the changed structures needs to be considered during ripple-effect analysis. This can reduce testing effort and even compilation time.

3. EXPERIMENTAL DESIGN

In the experiment, subjects were asked to design the data structures underlying an API. In doing so, subjects were faced with decisions such as the following:

- Which items should occur in the same structure definition?

- In what order should the fields be placed?

- What name and type should be chosen for a field?

In describing the design of the experiment, this section first describes the test instrument used, and then the four experiential hypothesis are considered. To begin with the test instrument provided each participant with instructions, followed by a page collecting demographic data, a practice question, and finally the three study questions. Each question included a specification of the domain specific information to be represented. The following is an excerpt of the instructions given to subjects:

> You will be asked to define some data structures to hold values for various kinds of information. This is not a race and there are no prizes for providing answers to all questions. If you do complete all the questions feel free to add any additional comments to your previous answers.
>
> Yumei is a medium sized island that gained independence in 1945. The island economy is based on the export of a various kinds of raw materials and agricultural products. The government has decided to publish a set of APIs that applications written for government departments must follow. ... The members may have any arithmetic type, or a pointer type, or an array type.

Subjects then performed the following steps.

- Read the information that needs to be represented.
- Decided which information should be held in a given data structure.
- Decided on the type and name of each field used to denote the information.
- Defined one or more structure types.

The three study questions related to the Department of Agriculture, the Department of Transport, and finally, the Department of Natural Resources. These three are referred to as the *Agriculture problem*, the *Transport problem*, and the *Natural Resource problem*.

Each question included variants that differed in two ways. First, the order in which items were listed in the specification was randomized for each subject and second the phrases used for the items were varied (several examples are given in Figure 3). In most cases (where reasonable alternatives existed) the phrases seen by a subject were randomly selected from two possibilities.

The experiment was run with two distinct populations of subjects. The first were professional programmers attending the ACCU [1] conference in 2005 and 2008, and the second, novice programmers who were undergraduate students at Loyola University. All subjects volunteered their time and were anonymous.

The collected data was then used to investigate two sets of hypotheses: the first deals with the relative order of the items in the specification and the fields in the data structure definitions. The second considers the clustering of fields into data structure definitions. Each set first considers overall trends and then compares the professional and novice programmers. This yields the study's four hypotheses.

Hypothesis 1 [Specification Order Influence]
The ordering of items in the specification strongly influences how subjects order fields within data structures.

Hypothesis 2 [Specification Order Comparison]
Novice programmers will be more likely to follow the order used in the specification (be less likely to be aware of a reason for using an order different from that provided in the specification).

Hypothesis 3 [Item Clustering]
Programmers will consistently cluster the same items together in structure definitions.

Hypothesis 4 [Item Clustering Comparison]
Professional developers are more consistent regarding clustering than novice developers.

4. RESULTS

This section reports results from the experiment as it relates to each of the four hypotheses. Overall 38 participants took part in the study. These included 26 professional programmers, having an average of 11.1 years of experience (standard deviation 6.3), who produced 35 complete answers (Agriculture: 24, Transport: 10, Natural Resources: 1). Answers that were incomplete at the end of the allotted time were dropped from the study. Subjects were told that they could use any computer language, and the languages used were (answer counts in parenthesis) C++ (18), C (13), Java (2), C# (1) and Python (1). Twelve novice programmers, students in their third and forth years of study, produced 26 complete answers (Agriculture: 11, Transport: 8, Natural Resources: 7) with five using C and 23 using Java.

When considering the individual structures, some subjects included fields that did not directly correspond to any item from the specification. Such fields were ignored in the following analysis.

4.1 Information Ordering

The presentation of the order-related data first considers Hypothesis 1 (Specification Order Influence) and then Hypothesis 2 (Specification Order Comparison). Analysis of the data requires understanding the number of possible orders, measuring the difference between two orders, and considering the chance that an order occurs randomly. To better understand these three, this section first considers an example data structure definition with four fields. In general, there are 24 possible orders that four fields can take. Thus there is a 1 in 24 probability of a particular field order occurring randomly.

A field order that differs from that of the specification's is evidence of external (non-specification related) influence. To quantify the difference between the order used in the specification and that found in the subject data structure definitions *field-order agreement* is used. This metric, which is computed per data structure definition, counts the number of pairs of fields appearing in the expected (i.e., specification) order. For instance, given a four item specification that uses the order 1234, the sequence 2314 is assigned the field-order agreement value four because 2 appears before 3 and 4, 3 appears before 4, and 1 appears before 4. To facilitate comparison, field-order agreement is often given as a percentage. The above example has a percent field-order agreement of 66% (4 of 6 possible pairs follow the specification order).

As can be seen in Table 1, **2314** (shown in bold) is one of five sequences that have a field-order agreement of four. This means that with probability 5/24, a random sequence of four fields will have a field-order agreement of four.

Turning to the real data, Figure 1 shows the percent field-order agreement (bullets) for the 133 structures created by subjects along with the probability that the particular order occurred randomly (crosses). To better understand the in-

Field-Order Agreement	Occurrences	Orders					
6	1	1234					
5	3	1243	1324	2134			
4	5	1342	1423	2143	**2314**	3124	
3	6	1432	2341	2413	3142	3214	4123
2	5	2431	3241	3412	4132	4213	
1	3	4231	4312	4321			
0	1	4321					

Table 1: Field-order agreement computed for all possible orderings of four fields where the specification uses the order 1234. The second column lists the frequency of each agreement value.

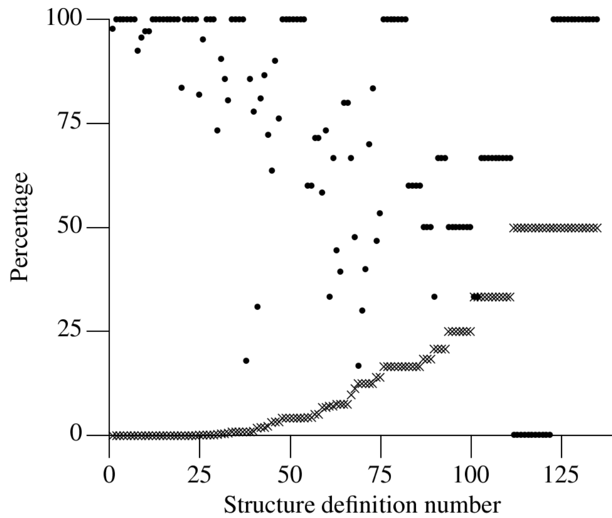

Figure 1: Field-order agreement and the probability of random occurrence. The x-axis includes each structure definition appearing in an answer ordered by the probability that their field-order agreement occurs randomly (crosses). Also shown is the field-order agreement (bullets). Both values are given as a percentage on the y-axis.

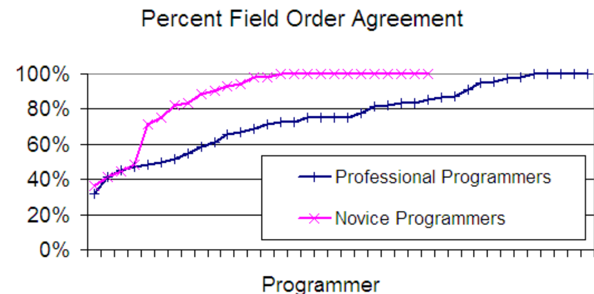

Figure 2: Percentage field-order agreement for every professional and novice programmer.

formation conveyed by these two marks, consider the example from Table 1. The order 2314 has a field-order agreement of 66% and thus would cause a bullet at 66%. It also has a probability that this field-order agreement occurs randomly of 20.8% (5/24) and thus would have a cross at 20.8%. The structures in Figure 1 are sorted along the x-axis by the randomness probability. Approximately the first third of the structures have very close to zero probably of occurring randomly. These crosses are from structure definitions containing the less likely combinations of six or more fields. Moving across the chart, by the right hand edge of the x-axis, the probably of a field order being randomly produced has risen to 50%. In other words, the crosses at the far right represent structures having two fields.

The dominate feature in the data associated with the percentage field-order agreement is that for structures having a very low probably of the order occurring randomly (those on the left of Figure 1), the percent field-order agreement is quite high. These are precisely the structures with a larger number of fields where such high agreement is unlikely. In fact, 25% have the same ordering as the specification and all but two have greater than 50% agreement with the specification. Had subjects randomly selected field order, agreement with the specification order would have an expected value of 50%. Thus, the field-order agreement is much greater than random agreement. In summary, the data summarized in Figure 1 supports Hypothesis 1.

Hypothesis 2 considers the impact that experience has on field-order agreement. Figure 2 shows the percentage field-order agreement for professional and novice programmers. The difference, visible in the graph, is statistically significant ($p = 0.015$) with professional programmers producing an average agreement of 75% (standard deviation 19%) and the novice programmers producing an average agreement of 86% (standard deviation 21%). Finally, at one extreme, considering only orderings that have less than 1% probability of occurring randomly, novices are twice as likely as professionals to exactly follow the ordering of the specification. Thus, there is support in the data for Hypothesis 2.

4.2 Information Clustering

Next, consider the two clustering hypotheses presented in Section 3. There are a large number of ways in which items can be clustered into a set of data structure definitions. The following analysis is patterned after the categorization analysis of Ross and Murphy who used a *Robinson matrix* to obtain information on the general categories created by their subjects [14]. A Robinson matrix has the property that the value of its matrix elements decrease, or stay the same, when moving away from the major diagonal (more mathematical details can be found in the Ross and Murphy article [14]).

V1 Date of last vet inspection
V2 Date animal born | Animal birth date
V3 Date crop sown | Crop sown date
V4 Date animal slaughtered
V5 Date crop harvested | Crop harvest date
V6 Antibiotics used | Antibiotics given
V7 Fertilizer used | Fertilizer applied
V8 Weed killer used | Weed killer sprayed
V9 Pedigree ID | Pedigree class
V10 Genetic ID | Genetic family
V11 Supplier of seeds | Seed supplier
V12 Was organically produced
V13 Number of each kind of animal on farm
V14 Farm acres used to grow crops
V15 Farm acres used to rear animals
V16 Average shipment weight
V17 Market value of kind of animal
V18 Market value of acre of crop
V19 Average labor cost for raising kind of animal
V20 Average labor cost for growing crop
V21 Feed-stuff costs | Cost of feed-stuff
V22 Fertilizer costs | Cost of fertilizer
V23 Location of field | Field location
V24 Location of farm | Farm location
V25 Farm owner | Owner of farm

Figure 3: The items in the Agriculture problem. (The alternative wording is listed where space allows.)

The Department of Agriculture problem is used in this analysis for two reasons. First, it produced the most answers (35) and second it contained the largest number of items (25), making it more likely that subject answers contain multiple structure definitions. The items are listed in Figure 3. The following two stage process was used to produce the Robinson matrix:

1. A similarity matrix was created from the structure definitions. The rows and columns of this matrix both represent the items listed in the problem, for instance if the entry for "Farm Owner" is in Row 3, then Column 3 also denotes "Farm Owner". The entries for the similarity matrix are calculated as follows: a) zero the matrix, b) for each structure definition add 1 to every entry in the matrix when two items, or rather their corresponding fields, are placed together in the same structure definition (e.g., if a definition contains fields representing both "Farm Owner" and "Farm Location" then add 1 to the entries at the corresponding row/column). Thus, the matrix is symmetric since two entries are incremented.

2. The original matrix is mapped to a dissimilarity form using the formula $1 - element/max_element$ for each *element* of the matrix, where *max_element* is the largest *element* value found in the matrix. Post mapping, all entries range from zero to one with zero representing the pair occurring together the most frequently. Using the dissimilarity form, the *seriate* function from R's [13] seriation package [8] was applied to produce a Robinson matrix. In the Robinson matrices shown in Figures 4, 5, and 6, values closer to zero appear darker.

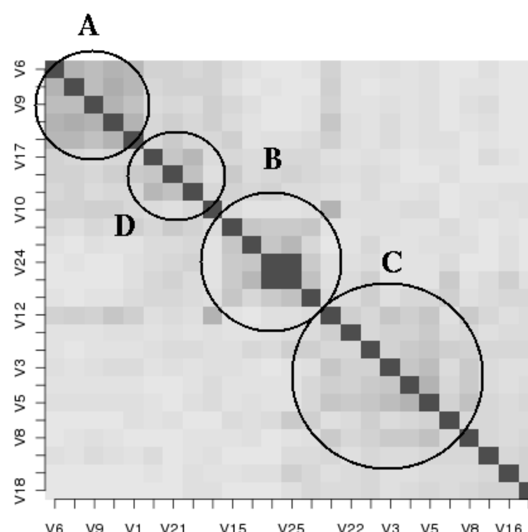

Figure 4: The Robinson matrix for the *all participants* field-order structure information. Darker areas indicate more closely associated fields. The order of items along the axes is 6 4 9 2 1 17 21 19 10 15 13 24 25 14 12 22 11 3 7 5 23 8 20 16 18.

The Robinson matrix produced using all subject data is shown in Figure 4. Several clusters are visually apparent. The four most prominent are highlighted by circles. Starting in the upper left, Cluster A (items V1, V2, V4, V6, V9) represents the *animals* found on a farm. Cluster D, the smaller 3x3 cluster adjacent to Cluster A (items V17, V19, and V21) represents the *cost of rearing animals*. Its proximity to Cluster A is caused by some but not all subjects placing the fields of Clusters A and D in a single structure definition.

Next, Cluster B (items V13, V14, V15, V24, and V25) contains the only two ubiquitously placed together Items V24 and V25 ("Location of farm" and "Farm owner"), which cause the 2x2 black square. The other three items account for the less dark region surrounding this black square. The name given to the definition containing this set of fields was often *farm* or something very similar.

Finally, Cluster C (items V3, V5, V7, V11, V12, V22, and, extending toward the lower right, perhaps V8 and V23) represent attributes of the *crops* raised on a farm. The association between items is weaker than with Clusters A and B. Also some items that might be expected to be part of this cluster (e.g., V8, "Weed killer used") have only limited association. In terms of Hypothesis 3, the existence of these clusters supports the hypothesis that programmers tend to consistently cluster similar items. A random clustering would result in a uniformly gray plot.

Finally, turning to Hypothesis 4, it is important to understand that the absolute order of the entries in a Robinson matrix is not important when comparing two Robinson ma-

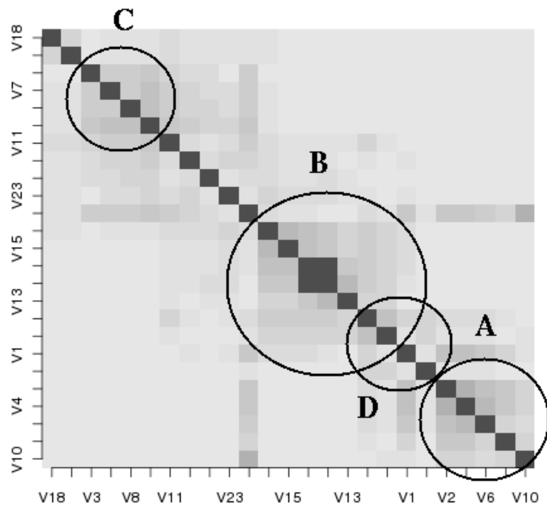

Figure 5: The Robinson matrix for the *professional programmers* item/field structure information. Darker areas indicate more closely associated fields. The order of items along the axes is 18 20 3 7 8 5 11 22 16 23 12 14 15 24 25 13 21 17 1 19 2 4 6 9 10.

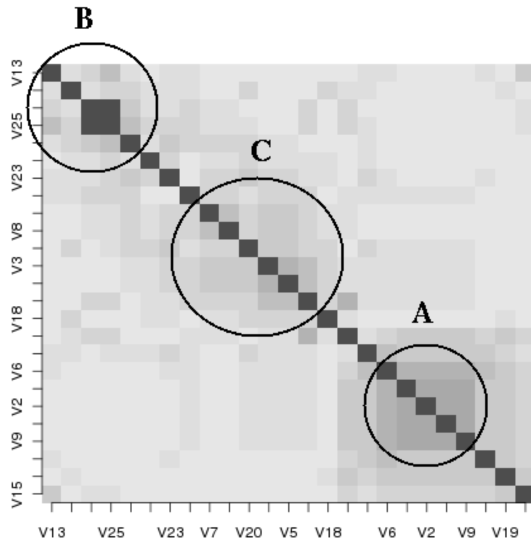

Figure 6: The Robinson matrix for the *novice programmers* item/field structure information. Darker areas indicate more closely associated fields. The order of items along the axes is 13 16 24 25 11 14 23 22 7 8 20 3 5 12 18 10 17 6 1 2 4 9 21 19 15.

trices. Only relative proximity is of interest. With this in mind, the matrices for the professional and novice programmers are shown in Figures 5 and 6, respectively. Before comparing the two, dominant structures of each are discussed.

Beginning with the professional programmer data (Figure 5), Clusters A, B, C, and D capture the same concepts as the clusters having the same label in Figure 4. However, this does not mean that they include exactly the same items. For example, Cluster A includes V2, V4, V5, V9, and V10 (swapping V10 for V1 when compared to Cluster A of Figure 4). V1 does appear two columns to the left and is related to the cluster just not as strongly (primarily because of an association to the items of Cluster B).

There is some visual evidence that professional programmers used a different clustering placement of some cost related items. Cluster B in Figure 5 includes at least two of the cost related items (previously identified exclusively with Cluster D) and Cluster A could also be expanded to include the costs (the tie is strongest for Item V1).

Next, Cluster C from Figure 5 has a more defined core, but could be extended to the lower right as there are weak associations there. This would make it closer in size to the similar cluster seen in Figure 4.

The final Robinson matrix (Figure 6) shows the data for the novice programmers. It has considerable similarity with the professional programmer data. The significant exception is that only three of the four dominant clusters are clearly present. Cluster D, related to costs, is absent from Figure 6.

In comparison with Figure 4, Cluster A of Figure 6 contains more items, while Cluster B contains fewer items. Also, Cluster C is slightly better defined.

When comparing the professional and novice programmers (Figures 5 and 6), Cluster A of Figure 6 contains more items, and Cluster B contains fewer items. Although Cluster C is larger, it is poorly defined by both groups. Finally, there is no obvious collection that might be called a Cluster D in Figure 6. An appreciation of costs is one area where novices would be expected to lag in experience when compared to professionals. Thus, this difference illustrates that categorization is at least partially driven by experience and provides evidence that the explanation-based theory of categorization is used by programmers.

A second difference comes from the placement of V1 (date of last vet appointment): V1 is clearly placed in Cluster A (animals) by the novices. In the professional data, V1 is in Cluster D (animal costs), but is not a great fit (there is considerable off-diagonal data). Combining the two data sets moves V1 out of Cluster D allowing Cluster D to become better defined (as seen in Figure 4). Further aiding this is a few of the novices associating costs not with the farm, but preferring to associate them with animals. Thus, Cluster D is more sharply defined and clearly associated with Cluster A in Figure 4.

Finally, the novice programmers appear to be more polar in their clustering. Said another way, the professional programmers produce more uniform answers. This provides evidence that experience drives decision making towards a smaller set of possibilities and is thus evidence that experience leads to greater uniformity. In other words, over time developers tend to think more alike.

This pattern is made apparent by considering two aspects of the Robinson matrix. First, Figure 6 shows considerable correlation off the main diagonal (in the lower left and upper

right of the chart). In comparison the professional programmers have zero correlation so far off the main diagonal. The only place they showed considerable disagreement is with Items V12 and V1. At the same time the novice programmers produced stronger "core" correlations. Thus when they agreed they agreed more strongly. This is difficult to see visually. One way to quantify this observation is to consider dissimilarity less than a chosen threshold. For example, consider the 10th percentile of the dissimilarities. Using all the data, the 10th percentile is 0.554, while for the novices and professionals these values are 0.545 and 0.612. Having the overall and novice numbers so close together indicates that the smaller dissimilarity in the overall data is dominated by the smaller novice population. A second view of this comes from counting the number of pairs having a dissimilarity less than a fixed value. For example, using 0.554 (the 10th percentile overall data), the novice programmers produced over three times an many such pairs (13 versus 4). Other similar thresholds produced a similar breakdown.

The final way of qualifying this difference is to consider the number of data structure definitions created to hold items from the specification. Considering just the agriculture questions, Figure 7 graphs the counts for both groups. While the visual evidence is not overly compelling, the differences is statistically significant $p = 0.041$. This data includes only structures into which a subject placed an item from the specification. Subjects created other structures, most often to describe a type. For example, a class Date that includes day, month, and year. Considering all definitions, the difference is even greater as p drops to 0.016. For both methods of counting, the following table summarizes the average number of structure definitions produced for each question and for all three questions taken together. Because the questions differ in the number of items they include, the final row describes the variation created by the question as well as by the subject.

Problem	Professional Counts Average	s^2	Novice Counts Average	s^2
Item-Representing Structures				
Agriculture	4.68	2.41	3.18	1.33
Transport	2.09	0.94	1.75	0.89
Resources	2.00	na	1.43	0.53
all data	3.79	2.36	2.27	1.28
All Structures				
Agriculture	5.46	2.50	3.64	1.36
Transport	2.70	1.25	1.88	1.13
Resources	2.00	na	1.43	1.27
all data	4.57	2.56	2.50	1.58

In summary, although not strong, there is support for Hypothesis 4 as the data indicates that the novices are more polar in their decision making.

5. RELATED WORK

The authors are not aware of any other work that has investigated decision making issues involved when developers create structure definitions. Various analysis of structure definitions in source code have been performed.

In general terms the clustering of items performed by Ross and Murphy study [14] (particularly Experiment 3) is very similar to this experiment. Their study did not compare the ordering of items in a cluster and against the order given

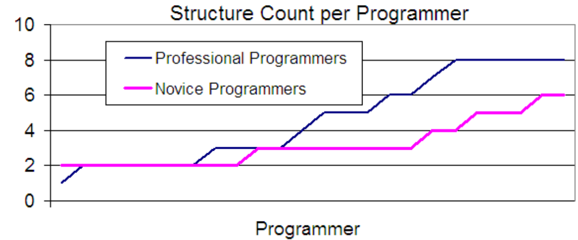

Figure 7: Struct-count comparison.

in the question. Rather it considered an experiment where subjects were given a list of various foods and asked to divide them into the groups: "similar food types," "foods that are eaten in the same situation," or "things that go together".

Jones [9] discusses various measurements of structure definitions in a large amount of C source code. The number of fields contained in the source code struct definition varied between 2 and 70 fields, with most definitions containing a small number of fields (a log-linear plot of the number of fields in a definition against the number of such definitions produced a straightish line).

Neamtiu, Foster, and Hicks [11] analyzed the release history of a number of large C programs, over 3-4 years and a total of 43 updated releases, and found that most structure definitions were unchanged between releases. Releases in which fields were added were more common than releases where a field was deleted (79% vs. 51%). A study of 3,600 Java classes by Dig and Johnson [6] also found that relatively few members were changed during the evolution of a project. A partial explanation comes from a study of C++ applications by Sweeney and Tip [15], which found that 11.6% of members were dead (i.e., never read from).

Finally, Anquetil and Lethbridge [2] analyzed 2 million lines of Pascal and found that record members that shared the same name were found to be much more likely to share the same type than members having different names.

6. THREATS TO VALIDITY

For the results of this experiment to have some applicability to developer performance, it is important that subjects work through problems at a rate similar to that used in their 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.

Developers often have the opportunity to interact with (e.g., ask questions of) people including the following: domain experts who wrote the specification, likely users of an API, and the customer. There was no such opportunity in this experiment; thus, subjects only had their own interpretation of the specification with which to work.

Developers often have the opportunity to study how information is going to be accessed by applications. This allows the data structures to be optimized for common access patterns or to make writing the code simpler. These design decisions are often made via an iterative process that may involve interaction with the code that uses them. Finally, the creation of data structures is usually an iterative process and the answers provided by subjects in this experiment can only be regarded as a first iteration.

7. SUMMARY

This study investigated factors that influence the decisions made when creating and maintaining data structure definitions. It also compared the differences in responses from novice and professional developers. The results show that

- factors exist that cause developers to create structure definitions that have a much smaller range of diversity than is theoretically possible,
- as a group, professional developers are more consistent in the items that they cluster together within a structure definition than novice developers,
- novice developers tend to cluster many more items into a smaller number of data structures, compared to professional developers, and
- the ordering of items in the specification has much more of an impact on novice developers than professional ones.

These results lay a foundation for the expansion of API guidelines [5] to consider clustering issues. Existing guidelines tend to focus on issues such as naming conventions, error handling techniques, default parameters, and which default constructors should be defined [5]. Thus, future work will consider approaches to improve the value of such guidelines. For instance, given that professional developers are more consistent when deciding which information should be clustered together, it makes sense that this task be carried out by professional rather than novice developers.

Finally, future work will consider the decisions that developers make regarding type and naming choices. For example, part-of-speech information will be examined in a search for patterns similar to those found by Caprile and Tonella in their analysis of function names [4].

8. ACKNOWLEDGEMENTS

The authors wishes to thank everybody who volunteered their time to take part in the experiments and ACCU for making a time slot available in which to run the experiment at the 2005 and 2008 conferences. Thanks to Matt Hearn, Brittany Babin, and Oluwaseyi Fatadi of Loyola University for decrypting subjects' hand written answers into machine readable form. Thanks to Michael Hahsler for suggestions on the use of his seriation package.

9. REFERENCES

[1] ACCU. Association of C and C++ users, 2009.

[2] 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.

[3] R. W. Bowdidge. Refectoring gcc using structure field access traces and concept analysis. In *Third International Workshop on Dynamic Analysis (WODA 2005)*, May 2005.

[4] B. Caprile and P. Tonella. Nomen est omen: analyzing the language of function identifiers. In *Working Conference on Reverse Engineering*, Altanta, Georgia, USA, Oct. 1999.

[5] K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison−Wesley, 2006.

[6] D. Dig and R. Johnson. How do APIs evolve?: A story of refactoring. *Journal of Software Maintenance and Evolution Research and Practice*, 18(2):87–103, 2006.

[7] G. Gigerenzer, P. Todd, and The ABC Research Group. *Simple Heuristics That Make Us Smart*. 2000.

[8] M. Hahsler, K. Hornik, and C. Buchta. Getting things in order: An introduction to the R package seriation. *Journal of Statistical Software*, 25(3), March 2008.

[9] D. M. Jones. Developer categorization of data structure fields (part 1 of 2). *CVu*, 20(6):14–18, 2009.

[10] G. Murphy. *The Big Book of Concepts*. MIT Press, 2002.

[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] E. M. Pothos and N. Chater. Rational categories. In *Twentieth Annual Conference of the Cognitive Science Society*, 1998.

[13] R Project. R, 2009.

[14] B. H. Ross and G. L. Murphy. Food for thought: Cross-classification and category organization in a complex real-world domain. *Cognitive Psychology*, 38, 1999.

[15] P. F. Sweeney and F. Tip. A study of dead data members in C++ applications. In *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, pages 324–332, June 1998.