An Empirical Study into the Effect of Data Flow Structure on Program Comprehension

Hong Yul Yang, Ewan Tempero
Department of Computer Science
The University of Auckland
Auckland, New Zealand
{hongyul|ewan}@cs.auckland.ac.nz

Abstract

Program comprehension is a critical part of software maintenance, an activity that bears a large part of the lifetime cost of a system. Studies suggest that program structure affects our ability to comprehend systems, with one such structure being data flow. We present the design and results of an empirical investigation into how a specific form of data flow can predict the effort of comprehension. The study consisted of two experiments, which were carried out on software engineering and computer science students at the University of Auckland. The results suggested that one of the major factors impacting comprehension effort was the effect of branching and interleaving data flow paths.

1 Introduction

The maintenance of software constitutes a significant part of the lifetime of a software system [18, p606]. If we are going to reduce the cost of maintenance, we must better understand the factors that affect this cost. Program comprehension is a critical part of any maintenance activity, and so we believe it is important to understand how aspects of code affect our ability to comprehend it. From the early days of principles of software development, program structure has been believed to influence our ability to comprehend software [23], however the exact relationship is not well understood. This paper presents an empirical study examining how a particular form of structure impacts comprehension.

It has been long thought that "complexity" is what makes things difficult to understand. For example, Banker et al. equate "software complexity" with "difficulty in understanding" [1]. But the actual relationship is difficult to determine, in part due to the uncertainty as to how to measure complexity. There are various views on what exactly constitutes complexity, the simplest of them being size, such as lines of code. More sophisticated views involve structure, such as control flow or data flow to capture the interaction between program components.

Different views of complexity are thought to have different effects on program comprehension. For instance, the Cyclomatic complexity number [14], which measures the number of distinct linearly independent paths through a function, is thought to indicate the effort involved in properly testing the function. On the other hand, spatial complexity [7], which is a measure of how "lexicographically" distant the program elements under comprehension are, is thought to indicate the effort involve in developing a mental model of the code.

We have previously introduced our notion of complexity based on data flow [21, 20]. We believe that this notion directly relates to certain aspects of program comprehension as many comprehension tasks involve following data flow paths. The question we have been trying to answer is exactly how does the *structure* of data flow relate to comprehension effort. The issue is that as our previous studies have shown there are many different ways to characterise the data flow, and each different way may impact comprehension differently (if at all).

We have developed *explanatory models* providing an explanation of how the structural characteristic of data flow affects the effort involved in a particular comprehension task. In this paper, we report on the first set of experiments in our research programme to determine which of our models best predict what happens in the real (empirical) world. Each experiment simulates several maintenance scenarios of following data flow paths to trace the source of the problem. The independent variables are the different structures of data flow paths, and the dependent variable is the effort required to trace the problem.

The rest of the paper is organised as follows. In the next section, we introduce the maintenance problem that moti-

vates our study and summarises related work. Our models relating data flow to comprehension effort is described in section 3. In section 4, we present our experimental design, including the hypothesis being tested, artifacts and procedures. We detail the results of the experiment in section 5 along with some statistical analyses. In section 6 we discuss the implications of the analyses and potential threats to their validity. Finally, section 7 presents our conclusions.

2 Background

2.1 Data flow maintenance problem

Consider the program in figure 1. At its current state, executing the program will lead to a failure within `D`. More specifically, a `NullPointerException` is thrown when it attempts to deference the value of the field `s` in the print statement at line 7, which turns out to be `null`. This is due to the fact that a `null` value was passed to `D`'s constructor, which is called from two places in the code: line 4 in `A` and line 3 in `F`. In both cases, the `null` value is from `C`'s field `str`, which is in turn set to the parameter of `C`'s constructor. The call to this constructor is in line 4 of `B`, and we see that its argument, `init` has not been initialised. This example should be a familiar one to all programmers.

In this example, we observe that `B`'s failure to initialise its field `init` caused a failure to occur at `D`. In other words, by changing `B` so that the problematic field is initialised to some value, we can stop the `NullPointerException` in `D`, which goes to show that `D` is dependent on `B`. Furthermore, the fact that there is no evident connection between `B` and `D`— for example, deleting or renaming `B` will not cause the compilation of `D` to fail (which implies that there is no compilation dependence, even transitively) — suggests that the dependency is non-trivial to detect.

This was in fact the motivation for studying *indirect coupling* [21, 20]. We have argued that coupling in the form of the above has an effect on maintenance effort, especially in understanding and modification. The above example is also a program comprehension task, as to complete it requires an understanding of all parts of the code that are involved in the data flow. To identify and trace these requires effort, and this led us to study the exact relationship (i.e. between data flow and effort) thereof.

2.2 Related work

Our study has been in examining how structure impacts program comprehension. While there have been many empirical studies related to program comprehension, most focus on what processes and techniques programmers use to understand code (e.g., [15, 11]). Of these, a particular study worth mentioning is Ricca et. al's investigation into the effect of the use of UML stereotypes on comprehensibility [16], which shares a similar experimental design approach and means of analysis with ours. There is also the study by Harrison et. al of how use of inheritance affects maintainability, which in their terms refers specifically to comprehensibility and modifiability [8]. Mathias et al. comment that many appear to pay little attention to structure and recommend that studies control for this attribute [13]. Much of the other work relating to program comprehension involves development of tools and support the comprehension process, such as software visualisation techniques, and often involve studies to determine their usefulness (e.g., [10]).

There has been a number of empirical studies on corroborating varying notions of complexity with comprehension effort. The aim of these studies were to investigate whether a given complexity measure is a valid predictor of comprehensibility. For instance, a study by Yang et. al [22] looks at the complexity profile graph metric, which quantifies aspects of control structure, and its effect on response time and error rate of maintenance tasks performed. Coupling is another closely relevant structural attribute that is frequently associated with comprehensibility, and there are a few studies specifically examining that association, for example, a study by Harrison et al. [9]. They examined two coupling metrics, CBO from the Chidamber and Kemerer suite [5], and their own Number of Associations metric. They found no relationship between either metric and comprehensibility.

Our notion of data flow complexity, along with the aforementioned spatial complexity [4, 7], share a common link that is the characterisation of *distance* and its association to comprehension effort. The distinction here is that in our case the distance is measured in terms of the number of elements along the data flow path.

Prior studies into the notion of dependence via data flow include “ripple effects” [3], which also deals with change propagation through data flow paths. Another closely related concept is program slicing [19], which is a technique for finding the parts of a program that can influence a given value at a specific point in the system. There have been several empirical studies trying to relate slicing with comprehension [2, 6]. However we find that these studies focus on *whether* slicing helps comprehension, and there is a lack of detailed insight into *how* slicing helps programmers understand structure in such a way that reduces the effort of comprehension. For this, we believe that we need to know how the structure associated with slices affects comprehension, and this is the line we are taking.

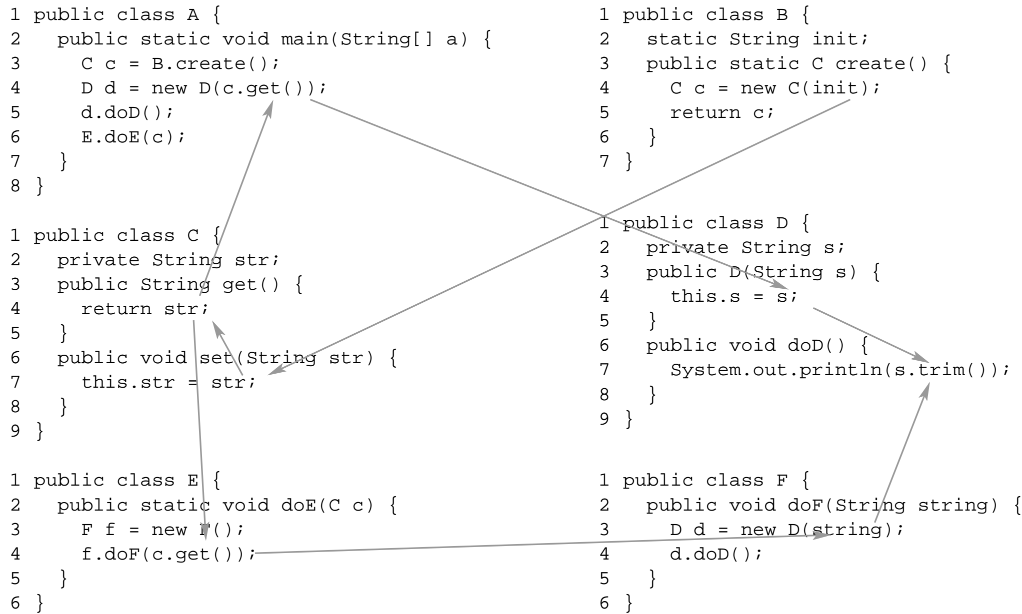

Figure 1. Null tracking example with superimposed data flow paths

3 Models relating data flow and effort

The type of comprehension problem presented in figure 1 is the focus of our study. Looking back at this program, we observe that `D` and `B` are connected by data flow *paths*. The print statement in `D` *uses* a value, that is *defined* in `B`. The arrows superimposed on the code indicate the individual usage-definition relationships between the two classes. We can express *comprehension effort* in terms of the cognitive load of tracing the chain for the `null` value used in `D`. This involves navigating between different statements, methods and classes that lie within the path. The fact that there are two different paths from `D` to `B` also gives rise to the question of whether it would be easier if there were just one single path, or conversely be harder if there were even more paths. Complex paths could incur cognitive load due to the need to keep track of different branches in the programmer's memory.

In [20] we proposed ways to quantify these data flow paths to reflect effort related to this form of comprehension. To do this we created several possible models, each associating a different attribute (essentially a measure) of data flow structure with comprehension effort. The structure is represented as a graph representing data flow of the coupling relationship, where edges correspond to individual usage-definition relationships between program elements, which are represented as nodes. Here, the program elements can either be statements, basic blocks, methods or classes, depending on the *granularity* involved. The concept of granularity is explained further on.

The following gives a definition for each model. In this definition, $Paths$ denotes a set of all data flow paths a given value can follow. $Nodes$ and $Edges$, respectively, are a set of nodes and a set of edges that lie within the paths in $Paths$. Note that the symbol $\propto$ used here denotes "association", not necessarily a linear correlation as it would mean in the strict sense. We do not want to commit to a prediction on the exact nature of each relationship until we know more.

1. $Effort \propto |Paths|$ — i.e. effort is associated with the number of paths (size of the set $Paths$)

2. $Effort \propto \sum_{p \in Paths} length(p)$ — i.e. effort is associated with the sum of the lengths (measured in number of edges) of all paths in the set $Edges$ This model contrasts the above in that it implies that paths are likely to induce more effort when they are longer.

3. $Effort \propto |Edges|$ — i.e. effort is associated with the size of the set $Edges$. This model contrasts the above in that having more and/or longer paths does not necessarily mean more effort. According to this model, if several paths largely share the same edges then they on the whole would not require as much effort as when they are more "dispersed".

4. $Effort \propto |Nodes|$ — i.e. effort is associated with the number of nodes (size of the set $Nodes$).

Granularity adds another layer of concern to the above models. Granularity determines the level at which the

counting should be done. For instance, consider the data flow path in figure 1, which goes through the methods `B.create()`, `C,set()`, `C.get()`, `A.main()`, the constructor of `D` and finally `D.doD()`. The length of this path would be 5 when counted at the *method* level, or 4 at the *class* level. Now take another data flow path, which goes through the same number of methods (5) as the above but all within a single class. Should we expect the effort required to trace these two paths to be the same because they have equal lengths at the method level, or expect the latter path to require less effort as it takes place within a single class? The only way to find out is to test the interaction between different granularity levels.

4 Experimental Design

Kitchenham et al. have recently made recommendations on the research and reporting processes for empirical studies in software engineering [12]. These recommendations have motivated our studies and guided our presentation here.

4.1 Hypotheses

Our goal is to determine whether (if any) one of the models predicts the empirical world better than the others. While we eventually would like to test all of the models, for the scope of this study we focus on a subset of these. We focus on two of the models described above – 1 and 3 – and also investigate the effects of granularity. For 1, we isolate the number of paths and ask whether comprehension effort truly varies according to varying numbers of paths. For 3, we isolate the number of edges and ask whether comprehension effort truly varies according to varying numbers of edges. For studying the effects of granularity, we focus on the interaction between two distinct levels: method-level and class-level. Here, we control the measures at the method level and ask whether varying the measures at the class level affects comprehension effort. We formulate these tests in terms of the following null hypotheses:

$H0_1$ (Effect of number of paths on effort) : Given two programs `A` and `B` where `A` has more paths than `B`, the effort required to comprehend `A` is no greater than that required to comprehend `B`.

$H0_2$ (Effect of number of edges on effort) : Given two programs `A` and `B`, where `A` has more edges than `B`, the effort required to comprehend `A` is no greater than that required to comprehend `B`.

$H0_3$ (Effect of granularity on effort) : Given two programs `A` and `B`, where `A` and `B` are equivalent in terms of the measures (paths, edges, etc.) at the *method* level, and yet any set of measures for `A` are greater than those for `B` at the *class* level; the effort required to comprehend `A` is no greater than that required for `B`.

4.2 Experiment arrangement

We ran two separate groups of experiments, where the first was to test $H0_1$ and $H0_2$ and the second was to test $H0_3$. This, as opposed to testing them in a single experiment, was done primarily due to practical considerations. We knew from initial trials that each experimental session on average would take around 1.5 to 2 hours, which was the upper bound on how much time the participants were willing to commit themselves to, given the voluntary nature of the experiment and the limits on the incentives we could provide them (each participant received a movie voucher).

The experiment participants were drawn from present and former students at the University of Auckland in the computer science and software engineering [17] programmes. They were mostly current students at their 2nd, 3rd and 4th years, but also included several graduate students and alumni working in industry. We advertised the experiment through mailing lists, department notice boards and announcements during lectures.

4.3 Artifacts and treatments

4.3.1 Programs

For the first experiment, we created four programs, designated Phi, Lambda, Sigma and Delta. We named them in this way in order to avoid possible bias. The structure of the programs are illustrated in figure 2, where nodes in the graph correspond to classes and edges correspond to data flow. We divided the programs into two pairs, each of which was a basis to test the appropriate null hypothesis: the pair Phi and Lambda is associated with $H0_1$, while the other pair Sigma and Delta is associated with $H0_2$. In the actual code, we inserted some auxiliary classes (apart from those implied in the figure) that did not contribute to the main structure shown above in order to control the size of the program as a whole. For each program there was a designated *"start" class*, which is where a particular value is used, and multiple *"end" classes*, which contain operations that eventually influence (define) this value. A fault was injected to one or more of the end classes such that it causes a failure on the start class, and this created the basis for the comprehension tasks described later on.

The programs for the second experiment had the same basic setup as those for the first. They were designated Beta, Nu, Epsilon and Omega, and their structures are presented in figure 3. Here, each node in a graph corresponds to a method, whereas an oval enclosing a group of nodes represents a class containing the corresponding methods. The

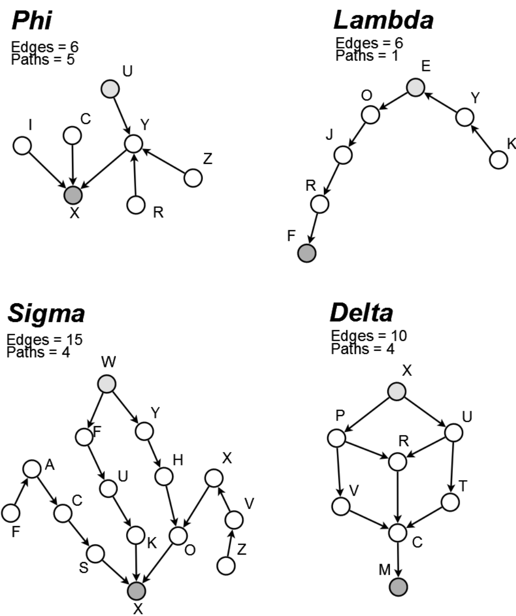

Figure 2. Data flow representation of the two pairs of structures under test for first experiment

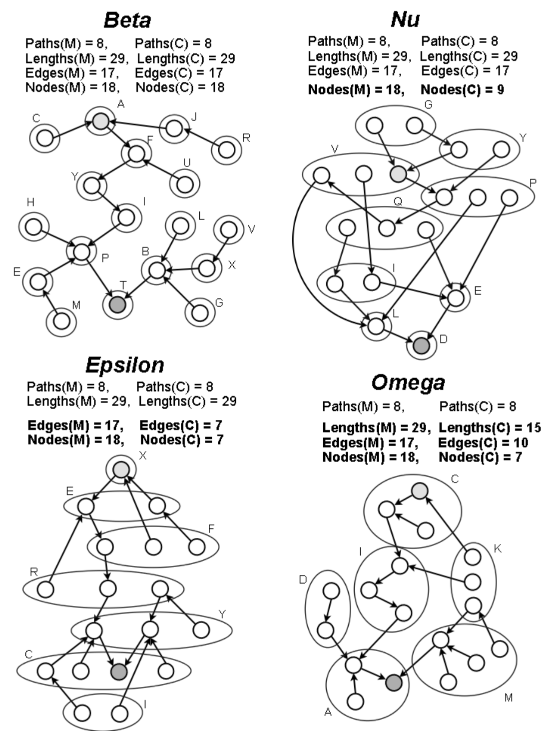

Figure 3. Data flow representation of the structures for second experiment: *measure*(M) and *measure*(C) denote the *measure* at method- and class- level granularity, respectively.

edges again represent data flow. In this experiment, each program represents a different means of coalescing methods into classes, and this is reflected by the differences between the measures at the method-level and those at the class-level. Notice that the data flow structures for all four programs are topologically equivalent at the method level. Furthermore, for each program, different sets of measures are varied across the two granularity levels, while at the same time the complementing sets of measures are equal at both levels. The program Beta is considered a baseline case, where all the measures for the program are equivalent at both method and class granularity levels. In Nu, only the number of nodes is varied across granularity, whereas in Epsilon, both numbers of nodes and edges are varied, and in Omega, all measures except for the number of chains are varied.

All programs were automatically generated from a description of the different models they correspond to. The description is in the form of a graph that details data flow and some basic control structures between classes. Figure 4 shows some examples of the classes that were generated. Here, class `X` is the "start class" where a failure occurs due to the value of `anza` being negative. Class `Z` is one of the classes that provide this value and depending on random factors the value could have been originated from either a return from calling `R.hiaticula()` or the parameter `edlu`.

It can be seen that there is no specific meaning to the names of classes, methods and variables used in the programs. The artificiality of the programs was a control measure so that we could focus solely on the structure of data and control flow. This also meant that the programs on the whole were not designed to perform any particular realistic task. We discuss the issue of artificiality in section 6.

4.3.2 Tasks

We created tasks to define the comprehension activities. A task was created for each of the programs under test and it required the participant to trace the location of the fault as described above. We employed a cross-over design where all participants had the complete set of treatments, without any indication or implication of how they relate to the hypothesis. One of the reasons for doing this was to utilise a small sample size. Another reason was it would allow for a more meaningful result as we were interested in the comparative difference within the individual, more so than the difference between the collective absolute times across individuals. The orders of the treatments however were randomised to avoid learning bias.

```
public class X {
  public static void rosenthal(int cech) {
    double anza;
    anza = Math.cos(Math.toRadians(cech));
    System.out.println(Math.sqrt(anza));
  }
}

public class Z {
  public static void richd(int edlu) {
    int mang;
    int rand = (int) (Math.random() * 2);
    switch (rand) {
    case 0:
      mang = R.hiaticula();
      break;
    case 1:
      mang = edlu;
      break;
    }
    X.rosenthal(mang);
  }
}
```

Figure 4. Example of generated code

In addition to the main tasks, two extra tasks were created, neither of which were to be factored in the analysis. The first task was the "primer" task, which required the participants to carry out an activity similar to that required by the main tasks, except on a small program with a more intuitive construction than that of the programs for the main tasks. The goal of the task was to familiarise the participants with the experimental setting. A second task was set up with a similar intent, but in this case the participants were unaware of the fact that it was a "dummy" task.

4.4 Procedure

The experiments were run across several sessions to accommodate for the availability of the participants. The participants were restricted to the use of the Eclipse IDE (integrated development environment). They were also required to use a web-based application, called TimeTracker, through which they were to record the time they took to solve each task.

Before the experiment began, the participants were asked to fill out a demographic questionnaire on their educational background, programming experience and programming ability (reflected by grades attained for courses with large programming components).

The experiment began with a briefing session that gave them an overview of the steps they have to follow and the environment they were restricted to. A brief tutorial was given to make sure that they were on even playing field with respect to the environment. The tutorial covered some useful navigational features in Eclipse (e.g. call hierarchy searching) and the use of the TimeTracker web application.

After the briefing, the participants were prompted to start on the primer task and asked to use the opportunity to familiarise themselves to the environment. On everyone's completion of the primer task, a debriefing session was given, which explained the correct solution. The necessity of this was prompted by trial experiments that were done to check the experimental design. They identified a problem that participants often gave an answer that indicated lack of understanding of the task, and so the debriefing was introduced to deal with this.

Following the debriefing, the participants were instructed to proceed with the actual tasks. For each task, they were to first read the description detailing which program the task concerns and what the problem is, start the timer as soon as they were about to begin comprehending the program, and then stop the timer as soon they have worked out their answers.

A short interview was carried out for each participant after they have finished all of the tasks to profile their comprehension methods and their perception on the relative difficulties of the tasks.

5 Results

5.1 First experiment

The first experiment had 22 participants over 6 sessions. For each of the 22 participants we collected the times they took to complete each task along with the correctness of their solutions. This information is summarised in table 1. For incorrect solutions, the times are shown in boldface italic. We see that on average (median), the participants took a longer period time on Phi than on Lambda. They also took a slightly longer period of time on Sigma than on Delta, but the difference does not appear to be as significant as that between Phi and Lambda.

The times for both Phi and Lambda are comparatively more widely spread than those for the Sigma and Delta pair. In particular, there is a notable outlier (participant number 6) within the elapsed times for Phi, which contributes considerably to the large standard deviation. Yet the time taken for that participant to complete the other three tasks was not out of line with that for the rest of the participants. A possible explanation for this is the fact that the particular participant was still adapting to the experimental setting.

We see that there were more incorrect solutions given to Phi than to Lambda. Also, there were more incorrect solutions given to Sigma than to Delta, which in fact everyone answered correctly. Factoring out the times for participants who gave incorrect solutions does not appear to affect the average. Only for Lambda is the average time for incorrect solutions notably different from the general average, but since there are only two it is difficult to say whether this is significant.

Participant	Phi	Lambda	Sigma	Delta
1	314	234	239	388
2	147	179	***537***	618
3	292	652	449	316
4	289	187	***285***	329
5	779	331	676	645
6	1541	306	256	349
7	846	383	449	344
8	***135***	227	340	193
9	645	370	527	572
10	***446***	221	790	390
11	***294***	***542***	***412***	478
12	***625***	631	377	190
13	375	412	388	583
14	260	147	***619***	479
15	351	143	370	527
16	***477***	131	***201***	123
17	***434***	179	600	469
18	***455***	270	***487***	279
19	***382***	347	683	279
20	357	145	534	666
21	716	260	393	464
22	377	***631***	373	555
Median	379.5	265	430.5	427
Median(cor.)	366	247	421	427
Median(incor.)	440	586.5	449.5	N/A

Table 1. Elapsed times in seconds for all tasks. Times in boldface italic correspond to incorrect solutions

Pairs	W	N	σ	z	p(1-t)
Phi vs Lambda	137	22	61.60	2.22	0.0134
Phi vs Lambda (C)	67	13	28.62	2.32	0.0101
P vs L*	115	21	57.54	1.99	0.0233
P vs L* (C)	54	12	25.50	2.10	0.0179
Sigma vs Delta	39	22	61.60	0.64	0.2607
Sigma vs Delta (C)	10	16	38.68	0.27	0.3930

Table 2. Wilcoxon signed-rank test for Phi-Lambda and Sigma-Delta pairs

Our goal is to see if the null hypotheses can be rejected. Specifically, we need an appropriate method that allows us to find out whether there is a significant difference between the times for a pair of programs. Since the distribution of the data is unknown, suggesting a non-parametric test, and we want to test pairs of correlated samples, we determined that the Wilcoxon signed-rank test was an appropriate method to apply to the data. In addition, the hypothesis implies a one-tailed test, as we are not only interested in the presence of a difference but also the direction thereof.

The results of the Wilcoxon test are summarised in table 2. The first column (**Phi vs Lambda**) shows the result of testing the times for Phi against those for Lambda over the whole sample. The following column shows the same test for a subset of the sample who have given the correct solutions to both Phi and Lambda. The next two columns are analogous to the previous two, except that they both discard the one outlier participant #6 (this was done in order to see how much impact it has on the resulting significance). The next column shows the result of testing the times for Sigma against those for Delta over the whole sample. The final column shows the test applied to a subset of the sample who have given the correct solutions to both. The row **W** in 2 shows the sum of the signed ranks according to the Wilcoxon test. The following rows show the sample size, standard deviation and z value respectively. The last column shows the p-value for a directional test.

According to the p-value of the first column, that is, test between Phi and Lambda over the whole population, we can reject the null hypothesis $H0_1$ comfortably beyond the 0.05 level. We observe that the significance actually increases if we factor out incorrect solutions. Also, discarding the aforementioned outlier participant still gives a significance that allows us to reject $H0_1$ at the 0.05 level. Furthermore, the positive signed-rank test suggests that Phi requires more effort (time) than Lambda. On the other hand, the test between Sigma and Delta did not yield a significant result. It still showed that Sigma had a higher rank sum than that of Delta, the difference was not great enough to allow us to reject the null hypothesis $H0_2$ (“Sigma does not require more effort than Delta”) at the appropriate level.

5.2 Second experiment

For the second experiment, there were 17 participants, some of which have participated in the first as well, over 3 sessions. The elapsed times for each task are shown in the left half of table 3. There is a slight but not significant variation in the median times across all tasks (ranging from 490s – 650s), with Omega having the lowest median, and Nu having the highest. On the other hand there was some variation as far as the individual times are concerned, with Epsilon exhibiting the greatest noticeable spread of values. There is at least one value that deviates notably far from the median in each task. In each of Beta and Nu, there is one such outlier (participants 12 and 3 respectively). In Epsilon, despite the three notable values over 1000 seconds, its rather broad distribution of the values tends to mask these outliers. Omega, meanwhile, has two such notable outliers.

The solutions to the tasks in the second experiment were

	Times				F-measures			
Participant	**Beta**	**Nu**	**Epsilon**	**Omega**	**Beta**	**Nu**	**Epsilon**	**Omega**
1	605	644	1380	744	1	0.5	1	1
2	713	662	684	741	0.8	1	1	0.67
3	815	1410	902	1094	0	0.5	0.4	1
4	717	693	291	445	1	1	1	1
5	601	486	235	404	1	1	1	1
6	543	922	352	494	1	1	1	1
7	516	901	639	895	1	0.67	0.8	0.67
8	453	206	229	466	1	1	0.5	1
9	737	666	1215	876	0.8	0.4	1	1
10	297	548	334	446	0	0.4	0	0.67
11	273	425	339	286	1	1	0.5	1
12	1179	620	993	414	0.4	0	0.67	0.33
13	394	649	590	330	1	1	0.67	1
14	428	407	1227	739	1	1	1	0.8
15	169	395	316	292	0	0.4	0.29	0.33
16	438	566	686	1314	0.4	1	0	0.8
17	549	738	577	500	1	1	1	0.67
Median	543	644	590	494	1	0.83	1	1

Table 3. Elapsed times (seconds) and F-measures for all tasks, experiment 2

more complex than those in the first. It naturally followed that the correctness of the participants' responses were also more diverse. Thus, instead of obtaining a binary measure (correct or incorrect) as with the first experiment, we computed precision and recall, which respectively correspond to the ratio of the number of correct answers given by the participant to the number of all given answers, and the ratio of the number of given correct answers to the number of all correct solutions. We then computed the F-measure by taking the harmonic mean of the two to gain an overall indication of "correctness", as shown in the right half of table 3. While it is difficult to say that one task was generally more "well-answered" than another, we can make some observations. Nu, despite having a highest number of completely correct (F-measures of 1) solutions, also has the lowest median for F-measure, giving the impression that it is generally not very "well-answered". Conversely, Omega, while not having the highest number of completely correct solutions, seemed to be consistently well-answered, with its highest median, highest minimum and least variation in the measures. Beta was also one of the better answered questions, where it had a highest number of completely correct solutions and a highest median. Epsilon, despite its highest median, had the least number of completely correct solutions and a lowest minimum.

Table 4 shows results of the Wilcoxon signed-ranked tests performed between Beta (the baseline) and the rest of the programs. While none of the tests indicated a small enough p-value to show statistical significance and thus allow us to reject $H0_3$ at a sufficient level, we can see that one pair is particular close, namely Beta and Omega. These two are at the two extreme ends of the granularity spectrum, which is to say, the measures at class-level granularity for Beta are collectively the highest, whereas the opposite is true for Omega. An interesting outcome of this test is that the rank sum (W) is negative, which suggests that the elapsed times for Beta are in general lower than that for Omega. In fact this is true for all other tests. This is contrary to the primary intuition that method co-location (which is the case for Omega) should simplify comprehension. A possible explanation for this is that configuring Beta to have one method per class produced a side-effect of actually making it simpler to comprehend.

Pairs	**W**	**N**	σ	**z**	**p(1-t)**
Beta vs. Nu	-39	17	42.25	-0.91	0.1811
Beta vs. Epsilon	-45	17	42.25	-1.05	0.1461
Beta vs. Omega	-63	17	42.25	-1.48	0.0695

Table 4. Wilcoxon signed-rank test for Beta

6 Discussion

6.1 First experiment

According to the above analysis, we are convinced that Lambda and Phi are different with respect to comprehension effort. Furthermore it shows that Phi required more

effort, at least in terms of time. This suggests that "breadth" (number of paths) is a more critical factor in determining effort than "length". On the other hand we found Sigma and Delta to be not so different with respect to comprehension. Generally Sigma had a tendency to require more effort than Delta (both according to median and directionality of the Wilcoxon test), but the difference was not significant. This is interesting in that according to post-experimental interviews, Delta caused more difficulty to at least half of the participants irrespective of the actual amount of time they spent. That is, regardless of how much time the participants spent, they found Delta to be harder since they had to hold a lot of information in their memory and also having to deal with many branching and interleaving paths increased the cognitive load on them. This prompts a further study into isolating the effects of branching paths on effort.

6.2 Second experiment

Primary analysis of the results suggests that the difference in method and class granularity does not have as much impact on comprehension time as we originally believed. Still one test (Beta vs. Omega) was promising, and leads to a possibility that a higher significance can be gained with a larger population. We are convinced that granularity had some effect on *correctness* of the comprehension tasks, and this leads to suggest that we need a more descriptive measure for comprehension effort to directly factor in both time and correctness. When specifically asked in the post-experimental interviews whether co-location of methods had any impact on understanding, the participants were mixed in their responses. In particular, those who did not feel a difference were found to have heavily relied on the navigational features in Eclipse, one such feature being method call hierarchy searching. This leads to an interesting question of whether the use of modern tools support has affected our results.

6.3 Threats to Validity

There are several issues that must be considered with assessing our results. One concern is the accuracy of the participants' answers. For example a participant could have spent a short amount of time on a task but given a wrong answer. We considered this by also analysing the results with the results with incorrect answers factored out. The results (for Lambda vs. Phi at least) were still significant.

A drawback from our cross-over design is the possibility of a carry-over effect where the experience of one treatment makes subsequent ones easier. We have tried to address this in two ways. One was to randomise the order in which the participants were to deal with the tasks. The other was to give them a "dummy" task — one that looked like the other tasks but was not actually part of the experiment. All participants did this task first. This was intended to reduce the risk of the participants disadvantaging themselves earlier on in the tasks due to unfamiliarity. When we asked the participants whether they felt a learning effect as they progressed, they uniformly answered to have felt only to a small extent, but not large enough to have a visible effect on their effort.

The participants were self-selected and most of them were from the software engineering programme, or had recently completed the programme. The responses to the demographic questionnaire indicated that about two thirds of the participants performed well academically (most grades were some kind of "A") with the remainder performing adequately (passed almost all courses). This is a fairly uniform group, and so this raises the question as to what degree our results would hold with other groups. Without studying other groups, particularly more experienced developers (which we hope to do), we cannot provide a definitive answer to this, although we note that our use of a cross-over design should reduce the effect due to variability of the participants.

The experiments were not double-blind, however we feel there is little risk in this. We masked the programs through our choice of names, and it is very unlikely any of the participants were familiar with the specifics of the models in our study. Furthermore, even if they were aware of the models, they would still need to take the necessary code navigation steps to solve the tasks successfully.

6.4 Drawbacks to Artificiality

As with most experimental studies, we had to compromise between making the experiment as realistic as possible and maximising its validity through control and restriction. By contriving the artifacts it was possible to gain control over the exact structure being tested. However there is the question of whether the structures used in the experiment reflect those in real life software at all. This is partially answered through our previous study [20] where we applied the metrics to actual applications. While there might not be systems that look like any one of the programs used in the experiment, we can find similar *differences* in the structure between parts of the code of the applications we have studied. Since the focus on the experiment is in the difference between structures, the fact that all structures are contrived in the same manner is believed to cancel out the issue of whether they are realistic. On a similar note, another question to ask is whether the strategies that the participants used for solving the tasks apply to comprehension on systems at a larger scale. This is more difficult to answer, and to do this it would help to have participants with substantial industrial experience to comment on the differences between real and contrived settings.

Although our current study, due to artificiality, may not have made a direct contribution to understanding the effects of data flow on comprehension of real software systems, we maintain that it provides a solid platform for designing future experiments.

7 Conclusion

In this paper we presented the design and results of two controlled experiments to study how a particular structure, a specific form of data flow, affects comprehension effort. Our approach to experimental design was a systematic one: define a set of models of the relationship between structure and effort that is descriptive enough to automatically derive experiment artifacts (programs). This approach has the advantages of control and efficiency, but we admit that its artificiality raises issues.

The result of the first experiment suggested that that the number of paths, or "breadth", increases cognitive load and thus has a more significant effect on effort than just the lengths of paths. Verbal feedback from the participants further supported this finding. This suggests that we refine our model to treat branching of paths more specially.

In the second experiment, we found that granularity does not seem to have as notable an impact on effort as expected. On one hand it leads us to wonder about the possibility that the validity of the experiment played the role in downplaying the significance. On the other hand we could just assume that granularity does not have much significance. We need further studies to determine our position.

Our eventual plan is to run the experiments on real software, based on the refined models. The results of our current experiments will allow us to make an informed choice on the software artifacts to evaluate, which we would not have been able to prior to the study.

References

[1] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, November 1993.

[2] D. Binkley, L. R. Raszewski, C. Smith, and M. Harman. An empirical study of amorphous slicing as a program comprehension support tool. In *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, page 161, Washington, DC, USA, 2000. IEEE Computer Society.

[3] S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13:263–279, 2001.

[4] J. K. Chhabra, K. K. Aggarwal, and Y. Singh. Measurement of object-oriented software spatial complexity. *Information and Software Technology*, 46:689–699, 2004.

[5] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA '91*, pages 197–211, New York, NY, USA, 1991. ACM Press.

[6] M. A. Francel and S. Rugaber. The value of slicing while debugging. *Science of Computer Programming*, 40:151–169, 2001.

[7] N. E. Gold, A. Mohan, and P. J. Layzell. Spatial complexity metrics: An investigation of utility. *IEEE Trans. Software Eng.*, 31(3):203–212, 2005.

[8] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *The Journal of Systems and Software*, 52:173–179, 2000.

[9] R. Harrison, S. J. Counsell, and R. V. Nithi. An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering: An International Journal*, 3(3):255–273, September 1998.

[10] D. Hendrix, J. C. II, and S. Maghsoodloo. The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Transactions on Software Engineering*, 28(5):463–477, 2002.

[11] A. Karahasanovic, A. K. Levine, and R. Thomas. Comprehension strategies and difficulties in maintaining object-oriented systems: an explorative study. *The Journal of Systems and Software*, 80(9):1541–1559, 2007.

[12] B. A. Kitchenham, S. L. Pfleeger, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[13] K. S. Mathias, J. H. C. II, T. D. Hendrix, and L. A. Barowski. The role of software measures and metrics in studies of program comprehension. In *ACM Southeast Regional Conference*, 1999.

[14] T. J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[15] V. Ramalingam and S. Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Seventh workshop on Empirical studies of programmers*, pages 124–139, 1997.

[16] F. Ricca, M. D. Penta, M. Torchiano, P. Tonella, and M. Ceccato. The role of experience and ability in comprehension tasks supported by UML stereotypes. In *ICSE'07*, 2007.

[17] Software Engineering. The university of auckland. http://www.se.auckland.ac.nz.

[18] I. Sommerville. *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[19] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[20] H. Y. Yang and E. Tempero. Measuring the strength of indirect coupling. In *Australian Software Engineering Conference*, 2007.

[21] H. Y. Yang, E. Tempero, and R. Berrigan. Detecting indirect coupling. In *Australian Software Engineering Conference*, 2005.

[22] J. Yang, T. D. Hendrix, K. H. Chang, and D. Umphress. An empirical validation of complexity profile graph. In *43th ACM Southeast Conference*, 2005.

[23] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979.