A Modeling Methodology for Empirically Studying User Behavior: The Case of UML Diagram Usage

Gay Costain, g.costain@auckland.ac.nz,
Ananth Srinivasan, a.srinivasan@auckland.ac.nz

Information Systems and Operations Management
University of Auckland Business School
Auckland, New Zealand

Abstract. The use of UML diagrams and associated methodologies in the development of software applications has, on the one hand been presented as a standard, while on the other hand has been criticized by empiricists who have actually studied its usage. In this paper, we describe a useful empirical method to analyze data about the nature, extent, and quality of cognitive support that the use of UML diagrams provides to a software developer. The data was collected and analyzed in a controlled experimental setup from both experienced and novice users. Our approach to analysing data in this study has the potential for wide applicability in empirical validation studies where focus on the process of usage is important.

Keywords: software development; unified modeling language (UML); cognition

1 Introduction, Background, and Research Aims

Does the Unified Modeling Language (UML) support the cognitive efforts of software developers? UML was made a standard by the Object Management Group (OMG) in November 1997 [20, 26] yet no empirical research has justified that choice. Some experimental work suggests that UML may in fact be a counter productive methodology for software developers (e.g. [40, 41]).

The authors of UML intended it to be a modeling language to support object-oriented (OO) analysis and design [4]. They believed that modeling is central to all activities leading up to the deployment of good software. A UML diagram may represent an abstraction of a program's source code solution, and the source code forms a textural model for the executable program. OO developers were found by survey [11, 19, 21] to strongly believe in the advantages of OO Software Development (OOSD) and even non-OO developers were found to have fairly positive perceptions. If users believe that OOSD is the most advantageous method for software development, it is important that a standard modeling language, devised to

aid that development, fulfils its promise. The software development industry was canvassed for input into the composition of the standard [20, 27], but no empirical research supported UML's creation.

There are other influences at play on software developers. Software development productivity for users of OO modeling tools may be affected by the user's previous experiences in the problem domain [1, 3, 32, 39, 43], type of user [1, 39, 43], user's experience of the OO paradigm [1, 7, 35], modeling notation and its use for abstracting models [32], and programming environment [7, 25]. As highlighted by ISO 9241, Part 11 (1998), usability must be judged in context. A standard language should be beneficial for a wide variety of users and contexts.

As a consequence of the preceding discussion, our main aim in this research is to investigate empirically if UML notation can supply cognitive support to software developers. If UML is found to provide that cognitive support, then this research may influence the opinions of software developers and encourage them to use UML. Some justification will have been found for UML's selection as a standard language.

2 Literature Review and Research Model Development

In this section we review the literature related to the cognitive steps involved in solving software development problems, which include program modification. The goal is to arrive at a model of cognitive processes to drive our empirical work that will address the main research questions.

2.1 Applying Cognitive Theory to Software Development

Anderson [2] proposed that both declarative memory (factual knowledge) and procedural memory (knowledge manifested in performance) may be activated in problem solving. He asserted that 'cognitive skills are realized by production rules'. Law [29] noted the popular belief that coding, comprehension and debugging of computer programs were facilitated through cognitive plan retrieval and recognition. For the software development environment Jefferies et al. [18] defined a design schema as 'the abstract knowledge about design and design processes, along with a set of procedures that implement these processes'. The authors believed that a goal of software design was to break down a problem into sub-problems and that the design schema was composed of both declarative and procedural knowledge that assisted to this end. During the design process a decision must be made as to which sub-problem to solve next, and then find a solution for it. Thus a goal must be identified for the sub-problem whose attainment may be achieved by pattern matching with memories of past events stored in long-term memory (LTM). A solution may be evoked from

LTM, may be derived from information acquired from the cognitive problem space, or inferred from the use of mental simulations [18, 33]. For OO development, experts require internal schemas representing information on a specific problem domain plus schemas dependent upon the targeted programming domain [7, 25]. In fact 'system design involves the integration of multiple knowledge domains - knowledge of the application domain, of software system architecture, of computer science, of software design methods, and so on' [14].

Kim et al. [25] viewed OO programs as sets of rules for solving groups of problems. In their framework, rule development may occur in either the problem or solution domain, consistent with Maher and Tang's [30] concept of co-evolutionary design. Rule and instance spaces as suggested by Simon and Lea [42] are included for each domain. Rules may be induced by evoking previously stored schemas, by deriving from knowledge gained from the current problem, or by inferring from simulations in the instance space (refer [16, 18, 42]).

2.2 Cognitive Differences Between Experts and Novices in Software Development

One significant aspect of the transformation from novice to expert in any domain of learning is the acquisition of problem-solving schemas. Problem-solving schemas are memory representations which embody knowledge based on past experiences with a particular type of problem. The process of constructing such a representation is also called schema learning. [22, p. 75]. Experts in software development can recognise and recall meaningful patterns when they see them, whereas the novice, lacking appropriate internal representations (IR), cannot [3, 15, 31, 38, 45]. However, experts are no better than novices when unfamiliar patterns are encountered [3].

2.3 External Representations (ER)

Several possible perspectives from which to evaluate a graphical ER exist. Scaife and Rogers [39] perceived the three aspects in their framework for explaining external cognition: *Computational off-loading* highlights cognitive benefits of graphical representations, *representation* relates to the representation's structural properties, and *graphical constraining* refers to possible processing mechanisms. Petre [36] believed that effective use of an ER requires purposeful perusal. Thus a graphical ER may support the problem-solver if the notation (representation) is conducive to modeling the real world of the problem; the model constrains what may be inferenced to prime essentials; the content provides a suitable abstraction of the problem for computational off-loading; and the layout aids perusal. From this we may conclude that much of the responsibility for the success of an ER lies with the modeler who

controls content and layout. We present the following framework for the representational system of a distributed cognitive task for solving a problem by an individual problem-solver. This framework serves as a guide for our research.

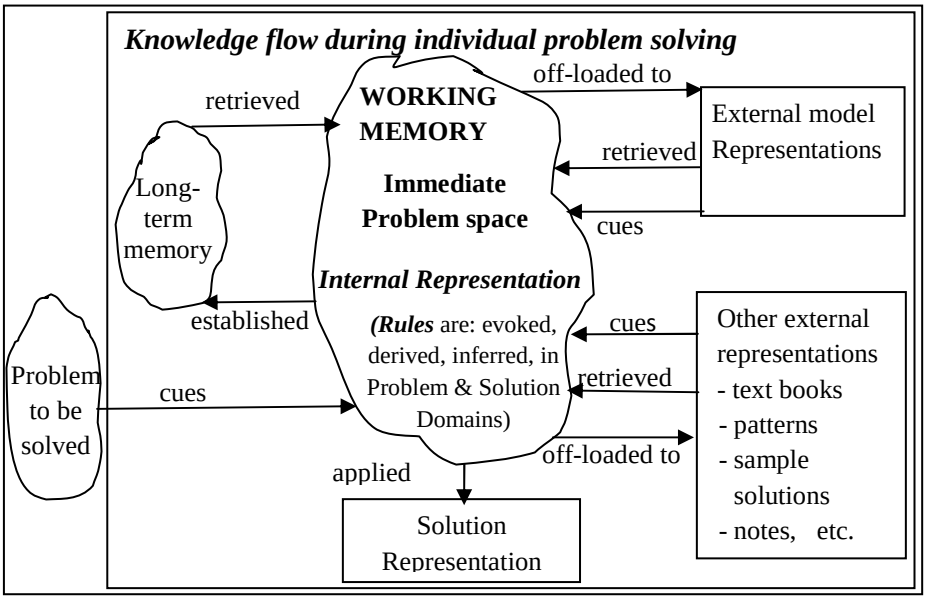

Fig. 1. Representational system showing knowledge flow for a distributed cognitive task for an individual problem-solver.

In Figure 1, the problem cues the formation within a problem space in working memory (WM) of an IR of the problem to be solved [33]. A plan/design schema may be retrieved from LTM to act as an executive structure for selecting and applying methods [7, 18, 29, 33, 37]. Rules may be derived, evoked or inferred in WM to aid comprehension of the problem, or to achieve sub-goals towards solution [25]. ERs may be utilised by the problem space [33, 46]. Knowledge retrieved from an ER may help establish the IR [39], may 'cue' some schema from LTM [34], or may be a recipient for the off-loading of chunks of data when short-term memory (working memory) becomes full [13, 23, 28, 33, 39].

An ER will be said to support a programmer's cognitive processes during software development if it does any of the following:

- aids comprehension of the problem by contributing to the IR of the problem in the problem space [12, 17, 34, 36].
- forms a set with the programmer's internal problem space to facilitate:

  - evoking of schemas in LTM [18, 25, 33].
  - deriving of rules from the problem space [18, 24, 25].
  - application of instances to simulate solution [15, 16, 18, 24, 25, 33, 42].
- provides a notation for off-loading of chunks to free WM [13 p.7, 23, 28, 33, 39].

3 Methodology

This paper is based on a study consisting of controlled experiments with experienced and novice modelers to obtain a rich understanding of the modeling process.

3.1 Controlled Experiment

Our intention was to empirically study the performance and behavior of modelers engaged in the modification of a non-trivial application with a view toward obtaining a rich picture of the process of modeling that occurs. While performance was an important part of the research, our emphasis in this paper is on examining *process behavior* during the modeling activity. In order to address this, the following research questions are examined:

- Can the use of UML documentation facilitate a developer's comprehension of a non-trivial program by assisting in the formation of valid IRs of a problem?
- Can the use of UML documentation facilitate a developer in modifying a non-trivial program by forming a set with the internal problem space?
- Can the use of UML documentation facilitate a developer's writing of code to solve a non-trivial problem by acting in a set with the problem space to induce rules for solution?
- Can the use of UML documentation facilitate a developer's modifying a non-trivial program by providing a vehicle for off-loading from working memory?

To study these issues, we conducted a controlled experiment with a group of 21 subjects, eight of whom had some industry experience. For the research reported in this paper, we examined in detail the activity of the five most successful modelers from each of the industry-experienced (expert) and student (novice) groups. The experiment involved the modification of two applications: invoicing and diary. Modifying a software program involves both the comprehension of the problem [5, 34] and the induction of rules to achieve the sub-goals that contribute to the full solution [25]. To study programmers' cognitive activities whilst modifying the programs, concurrent verbal protocols were collected for analysis. The use of verbal protocol analysis for process studies and their limitations are well discussed in the literature (e.g. [14, 44]). Since our focus in this research was on a deep understanding of process issues, we believed that the use of this methodology was appropriate.

Following a brief practice at talking aloud whilst programming, subjects were requested to modify each of the two computer applications written in the OO programming language VB.NET, only one of which was supplied with UML version 1.X use case, class, and sequence diagrams. The choice of diagrams reflected industry preference [8]. Use case descriptions were written in the format specified by Cockburn [6]. As per industry custom only the most important use case descriptions and sequence diagrams were included. The sequence diagrams documented the interactions for the use cases that were affected by the modifications.

The invoice modification required the addition of Goods and Services Tax (GST) calculations, the display of the calculated GST for each invoice item, and GST total, on the invoice form, and the inclusion of GST on the printed invoice. Not only was the calculation done for new items added to the invoice, but GST had to be adjusted when an invoice item was added to, or deleted. The diary application enabled users to record appointments with details. Its modification required that a set of one to three types of reminders be optionally added to an appointment. Whilst the application was running, on the day an appointment was due, and prior to the time of that appointment, reminder messages for the imminent appointments were to be triggered.

The sequence in which subjects carried out the modifications was rotated to ensure an even distribution of which application was encountered first and which was accompanied by UML documentation. The sequence of attendance at sessions depended upon the availability of the subjects. Paper, pencils and erasers were supplied. Subjects were advised that they could write on any supplied documentation.

Concurrent verbal protocols were collected whilst each participant modified each application. Ericsson and Simon [9] had concluded that simultaneous verbal protocols should not change the sequence or structure of problem solving, provided the subjects were not required to explain their actions as they performed. Transcribed protocols were analysed to find the sequence and category of the participants' cognitive steps during their modifications. Each modification ended when either the subject believed the task was complete or after two hours had elapsed, which ever came first.

4 Documenting Processes Using Behavior Graphs

The encoding categories for verbal protocol analysis should be clear and explicit, and should be defined prior to accepting input for encoding [10, 44]. The main goal of subjects is to modify an application. In order to achieve this, a number of sub-goals must be achieved. Sub-goals could include the acquiring of information related to the problem or solution, or the creating of a strategy to investigate or solve the problem. A goal may be achieved with the assistance of ERs such as UML diagrams, written text, online Help or the internet. The ER may be read by a subject or created by the subject during an episode (e.g. drawing a diagram or writing something). The

transcriptions of the subjects' verbal protocols were divided into episodes which were categorised as per Table 1, which encapsulates the criteria itemized in Section 2.3.

Table 1. Verbal protocol cognitive episode categories for solving a problem with the assistance of external representations.

Verbal Protocol Cognitive Episode Categories	
Problem Domain	***Solution Domain***
Form plan schema	Form design schema
Form internal representation	
Derive rule	Derive rule
Evoke rule	Evoke rule
Generate rule	Generate rule
Test	Test
Off-load	Off-load

A participant who remembers original code and returns to it in order to copy and/or modify it provides evidence that the code contributed to the modifier's IR. Checking a solution using instances may be carried out with the aid of an ER such as the code itself, or a diagram, and is categorised as 'Test'. When a participant creates an ER, reuses that ER, or modifies an existing ER, it is categorised as 'Off-load'.

Behavior graphs provide a method whereby each category of cognitive step in which UML documentation was involved may be quickly referenced. Each encoded episode is recorded vertically in the behavior graph in the sequence in which it occurred, within the column representing its cognitive episode category from Table 1.

Each episode is annotated with its sequence number, and with its source of cue or destination, and time duration, as shown in Figure 2.

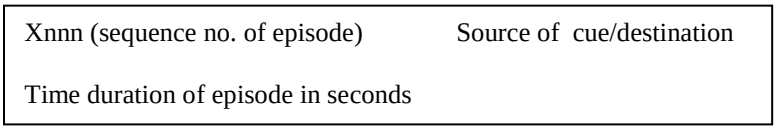

Fig. 2. Template for documentation of a categorised episode in a Behavior Graph.

X represents one of the six episode categories. The sequence number for episodes judged as creating an IR is preceded by an 'I', and, for episodes inducing rules, by an 'R'; other codes are: 'P' - plan schema formation, 'D' - design schema formation, 'G'

- generation, ‘T’ - testing, and ‘O’ - off-load. Any statement by a subject that could not be so categorised was deemed as ‘other’ and was not included in his/her graph.

5 Behavior Graph: An Example

In this section we provide one behavior graph example. Our intent is to show the value of using this technique in the study of detailed processes. The example was selected to display the richness with which we are able to capture details about the process. The behavior graph shown in Figure 3 captures the first quarter of the Diary modification protocol for Subject 16, the most experienced industry-based participant. The graph is helpful in showing a wide range of behaviors in which a participant may engage and that need to be accurately captured. The full graph spanning the entire protocol for this participant covers several pages; our objective here is to demonstrate the nature of the graph and the implications that can be drawn from it.

This subject completed the modification task in 1 hour 35 minutes. He was assessed as the most successful performer, achieving more sub goals than any other subject. We provide a commentary on how the behavior graph is interpreted.

5.1 Cognitive Episodes in Modification

- Subject 16 read the requirements (I1) and the use case documentation (I2) and as he made a number of informed comments about the diagrams it is assumed that he created IRs of the problem. He planned (P3) to check that the modification functionality was not specified in the documentation.
- He searched the class diagram to find where to put a reminder set, only to discover a ReminderSet class existed (I4). He checked the sequence diagram for the steps to add an appointment (I5), assuming that the existing ReminderSet required modification (G6). He reread the reminder set details in the specification related to the reminder set (I7) and generated the idea to add appointment details to the reminder sets collection (G8). (Note: Subject 16 later altered his theory.) He then studied the code behind the reminder set data entry form (R9).
- He discovered that the diary form was the start up form (R10), looked through the regions (R11), and then ran the application (T12). He theorised (G13) that the reminder sets be linked to the diary form. He added a reminder set check box to the diary form (R14), studied the sequence diagram (I15), then off-loaded his proposed changes into the ‘Add Appointment’ use case description (O16).
- In episode 17 he wrote the code for the check box, linking it to the reminder sets form. He unsuccessfully tested the code (T18), deleted it, studied the existing

code (R19), then reinstated the deleted code in another part of the application (R20). He successfully tested to bring up the reminder sets data entry form (T21).

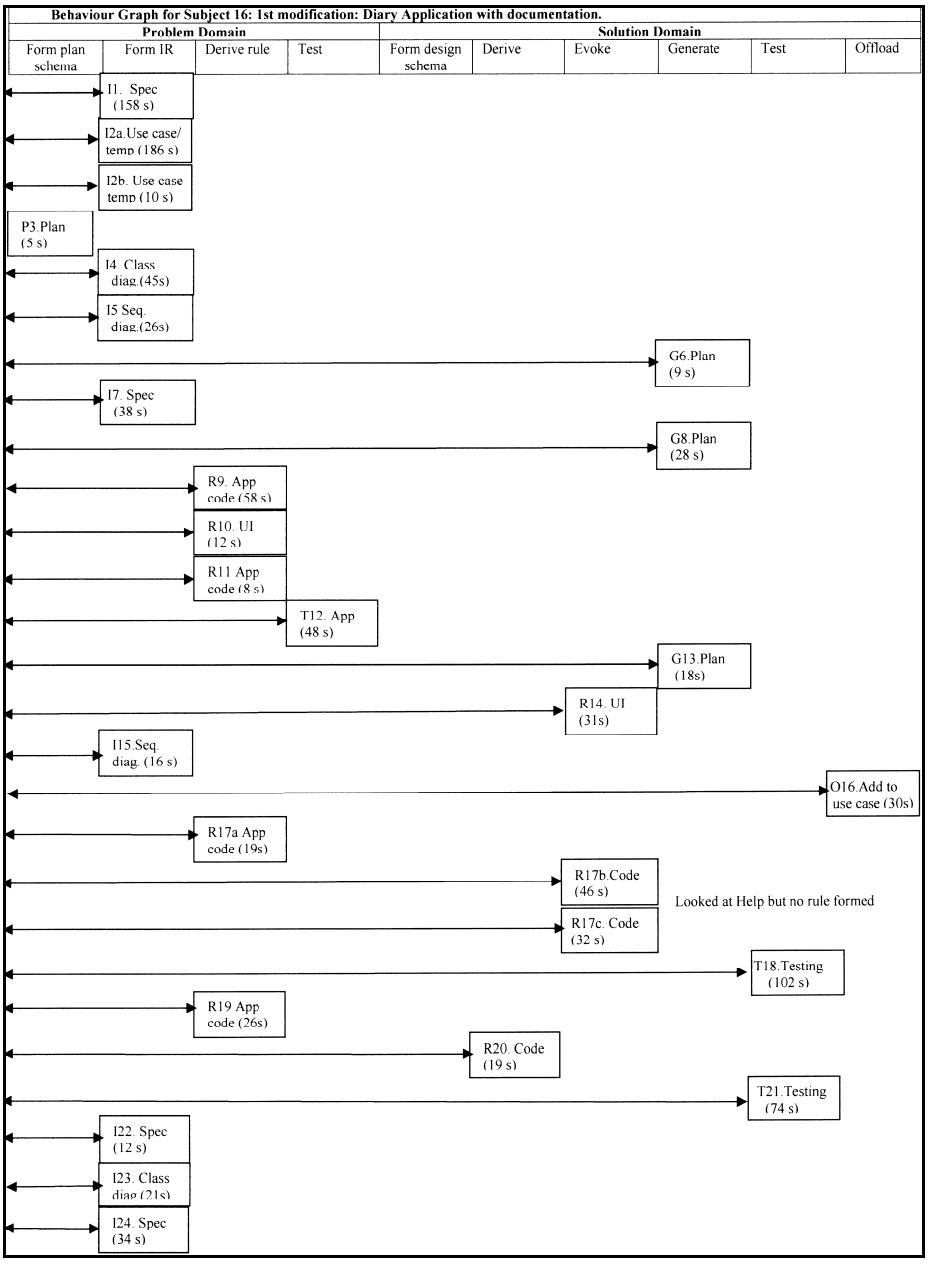

Fig. 3. Behavior Graph for Subject 16, page 1 of 2.

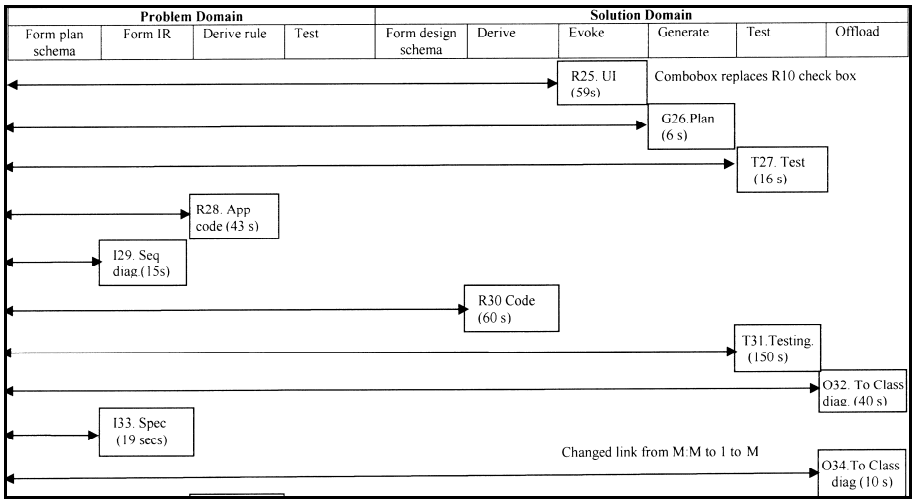

Fig. 3. Behavior Graph for Subject 16, Page 2 of 2.

- He read the requirements again (I22), then looked at the class diagram (I23).
- Subject 16 realised that his initial assumptions in G6 and G8 were wrong, and reconfirmed this by re-reading the specification (I24).
- He removed the checkbox, added a Reminders combo-box to the diary form (R25), and planned to load the combo-box (G26). He tested his code (T27), found he had forgotten to remove the code for the check box, and removed it.
- He studied the diary form loading procedures, found where to load the reminder sets (R28), then did not find reminder sets in the sequence diagram (I29)
- He copied the code for loading the reminder sets from the reminder set form into the diary form (R30), and successfully ran the program with his changes (T31).
- Subject 16 drew on the class diagram (O32), reread the requirements (I33) then changed his addition to the class diagram (O34) as a result of what he had read.

5.1.1 Subject 16 Forming an Internal Representation

The UML documentation appeared to assist Subject 16 in the creation of IRs. His comments indicated that reading the UML documentation (I2, I4, and I5) had clarified his understanding of the application. There is evidence that an IR was clarified as a result of his reading the class diagram (I4), when he discovered the ReminderSet class existed. He rejected his original plan to add appointment details to a reminder set after rereading the class diagram (I23) and changed his plan to add reminder details to

Appointment. He also used the sequence diagram to ascertain the functionality of the 'Add appointment' use case on several occasions: I5, I15, I29, R38, I72.

5.1.2 Subject 16 Off-loading to UML Documentation

In episode 14 Subject 16 added a check box to the diary form for selecting a reminder set. He studied the sequence diagram to find where to place his code to control the check box (I15), found the use case description more useful and off-loaded the information onto the use case description for 'Create Appointment' in O16.

Subject 16 also off-loaded changes onto the class diagram. He initially drew a joining class between the Appointment and ReminderSet classes (O32), re-read the requirements, changed his mind, removed the joining class and added a ReminderSet attribute to Appointment (O34).

5.2 Evidence from all Transcribed Protocols of Cognitive Support from UML

The results obtained for this research are based upon the analysis of collected, concurrent verbal protocols. The evidence that UML documentation provided cognitive support is based on the criteria itemized at the end of Section 2.3. Tables 2 and 3 summarize the results from the analysis of the transcribed protocols.

As can been seen from Tables 2 and 3, the class diagram was used productively in more episodes than use case or sequence diagrams. It is possible that the choice of application could affect the type of UML usage. The invoice application was process-oriented and should lend itself to process-oriented documentation such as use case, a fact taken advantage of by Subject 07 in his off-loading episode O9, and Subject 16 in his off-loading episode O16. The diary application was event-oriented.

Table 2. Cognitive support from UML documentation for NON-industry-experienced subjects.

Usage of UML in episodes with evidence of benefit achieved NON-Industry-experienced subjects										
		Aided comprehension				Used in set with problem space		Offloaded to provided UML documentation		*Drew new UML where none supplied*
Sub-ject	App	*Use case model*	*Use case descriptions*	*Class diagram*	*Sequence diagram*	*Use case descriptions*	*Class diagram*	*Use case descriptions*	*Class diagram*	
2	I			I3						
3	D			I15	R16	$I9^{P}$				
5	D	I36	I36	I39						O41, O43
7	I	I1	I2	I3				O9		
13	D			I6, I7b, I11, I41			$I13^{E}$, $R14d^{E}$, $I16^{P}$		O10, O26	

Episodes were entered into Tables 2 and 3 only if their content was found to be used in succeeding episodes. Thus the act of reading is not included if there was no evidence of its effect on further episodes. It is possible that some episodes that are not recorded here did contribute, but the evidence was not detected within the protocols.

The industry-experienced subjects exhibited more episodes on average of UML use. Few non-industry-experienced subjects had the experience (i.e. had the internal schemas) in software development to enable them to plan and progress swiftly through to solution.

Table 3. Cognitive support supplied by UML documentation for industry-experienced subjects.

Usage of UML in episodes with evidence of benefit achieved Industry-experienced subjects										
Sub-ject	App	Aided comprehension				Used in set with problem space		Offloaded to provided UML documentation		Drew new UML when none was supplied
		Use case model	*Use case descriptions*	*Class diagram*	*Sequence diagram*	*Use case descriptions*	*Class diagram*	*Use case descriptions*	*Class dia-gram*	
4	I						I19b^{P}, I22^{E}			
15	D		I2	I11, R14, I18, I22, I26		I2^{P}	R14^{DP}, I42b^{E}			
16	D	I2	I2	I4, I23	I29, R38, I72			O16	O32, O34	2 classes with IDs
17	I	I1		I2, I4, R42						Inherit-ance, attributes
20	I		I86	I4, I6, I16, I23a, I33, I72, I85, I97b			R45b^{D}, I45f^{D}, I49d^{E}, I49i^{E}, I49l^{E}, I61c^{E}			

[Note: Where UML documentation was used in a set to induce rules, the type of rule, whether derived (D) or evoked (E), is shown as a superscript following the episode number. A 'P' included in the superscript indicates problem domain, otherwise solution domain applies.]

5.2.1 UML Aids Comprehension in Formation of Internal Representations

The most popular use of UML documentation was in the formation of IRs (refer Table 4). The UML documentation was used to assist the subjects to familiarize themselves with the applications to be modified.

5.2.2 UML Documentation Forming a Set with the Problem Space

There was less evidence of subjects forming sets than there was for subjects forming IRs (refer Table 4). One explanation could be that, like Subject 16, subjects preferred to gain an understanding from the UML documentation, but resisted switching back and forth between screen and forms when coding commenced.

Table 4. Number of episodes providing evidence of UML cognitive support.

Number of episodes providing evidence of cognitive support								
	UML provides assistance to:							
		Form sets with problem space					Off-load from WM	
		Problem domain			Solution domain			
Subjects	Form internal representations	Derive rule	Evoke rule	Other	Derive rule	Evoke rule		Totals
Non-industry-experienced	13			2		2	5	22
Industry-experienced	26	1		2	2	6	5	42
Totals	39	1		4	2	8	10	64

5.2.3 UML Notation Used for Off-loading

Examples were obtained of subjects using the use case descriptions and class diagrams to assist in the development of what they intended to do. Steps for the planned changes could be added to use case descriptions (refer Table 2 Subject 07, episode 9, and Table 3 Subject 16, episode 16). The class diagram was used to develop the relationship between classes. Subject 16 off-loaded his thoughts for a relationship between the ReminderSet and Appointment classes in episode 32, which he rethought and corrected in episode 34. Subject 13 used the class diagram to the same end – adding an incorrect link in episode 10, then referring back to her link in episode 13, and finally correcting it in episode 26. These examples demonstrate the usefulness of external documentation in the planning stages for the modifications.

Off-loading that occurred when no UML documentation was provided often included rough sketches of relevant classes, the notation being simplified to suit the author. Subjects 05, 09, 16, 17 and 21 off-loaded sketches onto paper that communicated relevant class information but did not use formal UML notation. Where programmers create their own diagrams they may use any notation suited to their needs.

Two subjects drew non-UML diagrams to aid their understanding and plan what they would do. Subject 01 drew a flowchart and Subject 16 a structure diagram. Whereas Subject 01 was not a successful modifier, Subject 16 was the most

successful. He crossed off each section of his structure diagram as it was coded. His choice of notation reflected his familiarity with and prior use of structure diagrams.

6 Conclusion

From the results it was found that UML documentation did cognitively support programmers: it was found to assist in the creation of IRs of the problems, aiding comprehension; it was found to be used in a set with the programmers' problem spaces to assist in problem familiarisation and solution; and the UML provided a notation for off-loading from WM.

The number of subjects used in these experiments was small – twenty participants modified both applications, and ten of the more successful subjects had their verbal protocols transcribed and analysed to discover the cognitive support. The small sample size means that results cannot be applied globally to all programmers. However, it has been demonstrated that all transcribed subjects received cognitive support from the UML documentation. It has been demonstrated that UML can supply cognitive support. The question at the start of this paper has been answered.

The industry-experienced subjects, on average, used the UML documentation more than the non-industry-experienced subjects. There are several possible explanations for this. The industry-experienced subjects may have 'learned' to use the UML notation. The two best performing subjects worked in environments where class diagrams were used. Schemas for working with the diagrams may have been established that the non-industry-experienced subjects lacked. It is also possible the inexperience of the non-industry-experienced subjects inhibited their progress and reduced their opportunities to leverage the diagrams.

As a methodology to guide the study of processes using a modeling approach, we have found the use of behavior graphs to be particularly useful. It allows the researcher to extract, record, and analyze the full richness embedded in processes that have the potential of revealing details that might otherwise be missed.

References

1. Agarwal, R., De, P., Sinha, A.P., Tanniru, M.: On the Usability of OO Representations. Communications of the ACM 43(10), 83-89 (2000)
2. Anderson, J. R.: Rules of the Mind, Lawrence Erlbaum Associates, U.S.A. (1993)
3. Andriole, S., Adelman, L.: Cognitive Systems Engineering for User-Computer Interface Design, Prototyping, and Evaluation. Lawrence Erlbaum Associates, U.S.A. (1995)
4. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Massachusetts (1999)

5. Brooks, R.E.: Studying the Programmer behavior Experimentally: The Problems of Proper Methology. Communications of the ACM 23(4), 207-213 (1980)
6. Cockburn, A.: Using Goal-Based Use Cases. Journal of Object-Oriented Programming 10(7), 56-62. SIGS Publications, New York (1997)
7. Détienne, F.: Design Strategies and Knowledge in Object-Oriented Programming: Effects of Experience. Human-Computer Interaction 10, 129-169 (1995)
8. Dobing, B., Parsons, J.: Dimensions of UML diagram Use: A Survey of Practitioners. Journal of Database Management 19(1), 1-18 (2008)
9. Ericsson, K.A., & Simon, H.A.: Protocol analysis: Verbal reports as data. Cambridge, MA: MIT Press (1984)
10. Ericsson, K.A., & Simon, H.A.: Protocol analysis: Verbal reports as data. Rev. ed. Cambridge, MA: MIT Press (1993).
11. Fedorowicz, J., Villeneuve, A.: Surveying object technology usage and benefits: A test of conventional wisdom. Information & Management 35, 331-344 (1999)
12. Goel, V.: Sketches of Thought. A Bradford Book. MIT Press, Cambridge, Massachusetts (1995)
13. Grogono, P., Nelson, S.H.: Problem Solving & Computer Programming. Addison-Wesley, U.S.A. (1982)
14. Guindon, R.: Knowledge exploited by experts during software system design. International Journal of Man-Machine Studies 33(3), 279-304 (1990)
15. Guindon, R., Curtis, B.: Control of Cognitive Processes During Software Design: What Tools are Needed. Proceedings of the CHI'88 Conference on Human Factors in Computer Systems, New York. Pp.263-268. ACM (1988)
16. Haverty, L.A., Koedinger, K.R., Klahr, D., Alibali, M.W.: Solving Inductive Reasoning Problems in Mathematics: Not-so-Trivial PURSUIT. Cognitive Science 24(2), 249-298 (2000)
17. Hungerford, B.C.: Reviewing Software Diagrams: A Cognitive Study. IEEE Transactions on Software Engineering 30(2), 82-96 (2004)
18. Jefferies, R., Turner, A.A., Polson, P.G., Atwood, M.E.: The Processes Involved in Designing Software. In Anderson, J.R. (ed) Cognitive Skills and their Acquisition, pp.255-283. Lawrence Erlbaum Associates, Hilldale, New Jersey (1981)
19. Johnson, R.A.: The Ups and Downs of Object-Oriented Systems Development. Communications of the ACM 43(10), 69-73 (2000)
20. Johnson, R.A.: Object-oriented analysis and design – What does the research say? Journal of Computer Information Systems 42(3), 11-15 (2002)
21. Johnson, R. A., Hardgrave, B.C.: Object-oriented methods: current practices and attitudes, The Journal of Systems and Software 48, 5-12 (1999)
22. Kahney, H.: Problem Solving: Current issues. Second Edition. Open University Press, Buckingham, Philadelphia (1993)
23. Kim, J., Lerch, J.F.: Towards a Model of Cognitive Process in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies. In Bauersfield, P., Bennett, J., Lynch, G. (eds) Proceedings of CHI '92, ACM Conference on Human Factors in Computing Systems, May 3-7, Monterey, California, pp. 489-498 (1992)
24. Kim, J., Lerch, J.F.: Why is Programming (sometimes) So Difficult? Programming as Scientific discovery in Multiple Problem Spaces. Information Systems Research 8(1), 25-50 (1997)

25. Kim, J., Lerch, J.F., Simon, H.A.: Internal Representation and Rule Development in Object-Oriented Design. ACM Transactions on Computer-Human Interaction 2(4), 357-390 (1995)
26. Kobryn, C.: UML 2001: A Standardization Odyssey. Communications of the ACM 42(10), 29-37 (1999)
27. Kobryn, C.: Will UML 2.0 be Agile or Awkward? Communications of the ACM 45(1), 107-110 (2002)
28. Kotovsky, K., Hayes, J.R., Simon, H.A.: Why are Some Problems Hard? Evidence from Tower of Hanoi. Cognitive Psychology 17, 248-294 (1985)
29. Law, L.: A situated cognition view about the effects of planning and authorship on computer program debugging. Behaviour & Information Technology 17(6), 325-337 (1998)
30. Maher, M.L., Tang, H.: Co-evolution as a Computational and Cognitive Model of Design. Research in Engineering Design 14, 47-63
31. McKeithen, K.B., Reitman, J., Rueter, H.H., Hirtle, S.C.: Knowledge Organization and Skill Differences in Computer Programmers. Cognitive Psychology 13, 307-325. Academic Press (1981)
32. Nielsen, J.: Usability Engineering. Academic Press, U.S.A. (1993)
33. Newell, A., Simon, H.A.: Human Problem Solving. Prentice-Hall, U.S.A. (1972)
34. Pennington, N.: Stimulus Structures and Mental Representations in Expert Comprehension of computer Programs. Cognitive Psychology 19, 295-341 (1987)
35. Pennington, N., Lee, A.Y., Rehder, B.: Cognitive Activities and Levels of Abstraction in Procedural and Object-Oriented Design. Human-Computer Interaction 10, 171-226 (1995)
36. Petre, M.: Why looking isn't always seeing: Readership skills and graph. Communications of the ACM 38(6), pp.33-44 (1995)
37. Rist, R.S., Schema Creation in Programming. Cognitive Science 13, 389-414 (1989)
38. Rosson, M. B., Alpert, S. R.: The Cognitive Consequences of Object-Oriented Design. Human-Computer Interaction 5, 345-379, (1990)
39. Scaife, M., Rogers, Y.: External cognition: how do graphical representations work? International Journal of Human-Computer Studies 45, 185-213 (1996)
40. Shanks, G., Tansley, E., Nuredini, J., Tobin, D., Weber, R.: Representing Part-Whole Relationships in Conceptual Modeling: An Empirical Evaluation. Proceedings of the 23rd International Conference on Information Systems, Barcelona, pp. 89-100 (2002)
41. Siau, K., Cao, Q.: Unified Modeling Language (UML) – A Complexity Analysis. Journal of Database Management 12(1), 26–34. Idea Group Publishing (2001)
42. Simon, H. A. & Lea G.: Problem Solving and Rule Induction. In: Simon, H. A., (ed.) Models of Thought, pp. 329-346. University Press, London (1979)
43. Tabachneck-Schijf, H. J. M., Leonardo, A. M., Simon, H. A.: CaMeRa: A Computational Model of Multiple Representations. Cognitive Science 21(3), pp. 305-350 (1997)
44. Todd, P., Benbasat, I.: Process Tracing Methods in Decision Support Systems: Exploring the Black Box. Management Information Systems (MIS) Quarterly 11(4), 493-512
45. Wærn, Y.: Cognitive Aspects of Computer Supported Tasks. John Wiley and Sons, Essex, Great Britain (1989)
46. Zhang, J., Norman, D. A.: Representations in Distributed Cognitive Tasks. Cognitive Science 18, 87-122 (1994)