A Loop is a Compression

Walter Milner

University of Birmingham, Weoley Park Road, Selly Oak, Birmingham B29 6LL w.w.milner@bham.ac.uk

Abstract. An outcome from a larger research project is described. This places work seeking to understand how novices learn elementary programming notions in a wider framework derived from cognitive science, and in particular the group of ideas centered on conceptual blends. The framework is outlined and the research methodology is described, followed by some of the data gathered. It is suggested that students' consideration of code fragments can be analysed in terms of mental spaces, and that loop statements represent compressions. The implications for teaching are discussed, and future work is outlined.

1 Introduction

This work-in-progress report proposes a way of looking at students' understanding of programs which makes use of a set of ideas from cognitive science which includes frames, conceptual blends and compressions. In particular it argues that a loop is not literally a loop, but that statements in a loop are actually compressions, as described below, of statements in the unrolled loop.

The paper describes the background to the current work, and argues for the need to adopt a non-literal view of meaning. It provides an overview of frames and related ideas in general terms, and suggests a way of looking at program comprehension using these ideas. This is followed by a description of some interviews with students, and an exploration of the implications of this way of looking at concept development for teaching methods and course design. Finally there is an outline of where future work will proceed.

2 Background

This is concerned with how students with limited or no prior knowledge deal with their first exposure to some ideas in computer programming. It is part of some work concerned with understanding concept development in learning object-oriented programming (OOP) in Java, and in a wider context to develop this within a model of concept development within 'scientific' disciplines such as mathematics and physics.

There have been several reports of students' problems developing an understanding of OOP in Java (for example Biddle and Tempero [1], Ragonis and Ben-Ari [2], Eckerdal and Thuné [3], Griffiths and Woodman [4], Fleury [5]). While it is clear that we (that is, educators) have a clear understanding of these ideas of computer science, we do not have an understanding of that understanding, and similarly cannot explain the lack of understanding among some students.

While OOP is the final concern, this paper describes some work concerned with basic control constructs and simple variables.

3 Nonliteral Meaning

This section argues that program code, and discourse about programs, contains nonliteral meaning, despite the fact that program code appears to be a good example of literal meaning.

The concern is with the meaning of ideas, and in this context 'idea' is taken to mean a syllabus item, such as 'array' or 'function' or 'abstraction'. There is a paradox - we tell students what these ideas mean, yet some of them still do not know. For example, we tell them what 'array' means, but some of them cannot use them or understand programs which use them. How do we resolve the paradox?

The solution offered is that meaning is not a simple issue, and that a literal Objectivist idea of meaning is not appropriate.

Firstly, there is an implication that a distinction can be drawn between the meaning of an idea and an idea itself. For the purposes of this paper, meaning is taken to be a subjective interpretation. For example the meaning of 'array' is what a student says they think the meaning is. In a class there will be variations in the meaning given to the term. Among faculty staff there is likely to be much smaller variation. If we want to know the 'real' meaning of arrays, taken to be 'the idea itself', we can take it to be the normative view shared by faculty staff. This corresponds to a phenomenographic approach, as developed by Marton [6] and used for example by Eckerdal and Thuné [3] and by Booth [7]

Johnson [8] describes an Objectivist idea of meaning:

> Meaning is an abstract relation between symbolic representations (either words or mental representations) and objective (i.e. mind-independent) reality. These symbols get their meanings solely by virtue of their capacity to correspond to things, properties, and relations existing objectively "in the world".

and then contrasts this (page 5) with the way people understand each other:

> .. meaning typically involves nonliteral (figurative) cognitive structures that are irreducibly tied up with the conceptual or propositional contents attended to exclusively in Objectivist semantics.

A computer program appears to be an excellent example of Objectivist, literal meaning. Perhaps the best instance of this would be code generated by a Java GUI builder - the programmer 'draws' the user interface with labels and buttons and so on, and the GUI builder generates the code required to produce this. This means code is both generated and executed by the computer, and human understanding is bypassed.

It is therefore very tempting to extend this to asserting that discourse *about* program code can have its meaning analysed from an Objectivist perspective. In other words that the contents of a student textbook, or what is said in a lecture, 'means what it says'. But this leads to the above paradox, that some students do not understand some aspects of programming, even though they have been told all about it. A good example is given by Fleury [9] where a student is describing his reactions to a program which has been altered so that the data members are public and there are no accessor methods - he is asked if this is an improvement:

> Millions of times, I've seen an accessor, where it just does nothing but return a value. And I always thought in my head that that was just kind of goofy, so I really want to say better. But because of the fact that I've been kind of led to believe that that's not better, I'm not sure what to say.

The paradox is resolved by the realisation that much of the discourse about programming is nonliteral - it does not literally refer to what it says. However ordinary discourse about computing is also figurative, conventionalised so deeply as to make it difficult to recognise. For example, computers do not actually have memory. Memory is a mental characteristic of humans and other animals providing for the recall of past experiences, emotions and events. Digital systems have circuitry which is only metaphorically described as memory. Douce [10] gives many more examples of metaphor use in software development.

The focus of this paper is that a loop is not literally a loop, but that statements in a loop are actually compressions, as described below, of statements in the unrolled loop.

4 Conceptual Integration Networks

The idea of a compression is part of the notion of a conceptual integration network, and what follows is an attempt to give a brief outline of this set of related ideas. A key work in this area is 'The way we think' [11]. Conceptual integration networks are described in [12], and compressions are described in [13]. These ideas have been applied to mathematics [14], [15] and [16], and human-computer interface (HCI) [17]. Veale and O'Donoghue [18] consider the computational requirements of blending.

4.1 Frames

The term 'frame' is related to that of 'schema', which has been used by several psychologists to denote variations on the theme of a structured mental representation. Piaget [19] uses the term scheme for patterns of activity in infants in what he calls the sensori-motor stage, and Bartlett [20] demonstrated the role of schemata in memory and recall.

Minsky [21] uses the term frame as follows:

> When one encounters a new situation (or makes a substantial change in one's view of the present problem) one selects from memory a structure called a Frame. This is a remembered framework to be adapted to fit reality by changing details as necessary.
>
> A frame is a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is what to do if these expectations are not confirmed.

This is Minsky's meaning of 'frame', written towards the start of the development of artificial intelligence (AI), when human thought and computing were seen to have a simple correspondence. Consequently when he says 'a frame is a data-structure' he is implying that human cognition is appropriately seen in terms of data in computer memory. This is only loosely related to the way frame is used here.

Related to this is the idea of a script by Schank and Abelson [22], who describe the 'restaurant script' describing what people do when they eat out, as a way of structuring knowledge of the term 'restaurant'. This is the sense in which Rist [23] uses the term schema when he considers how programmers learn to develop plans to solve programming problems.

The term frame is used here in the sense of frame semantics, primarily derived from Fillmore [24], and is the idea that meaning depends on the context of the communication. The most commonly quoted example is the COMMERCIAL EVENT frame, which has slots including BUYER, SELLER, GOODS and MONEY. Knowledge of this frame means that

> John bought the car for a good price

delivers the meaning that the price was low, while

> John sold the car for a good price

means the price was high. The meaning of 'good' depends on knowledge of the BUYER and SELLER roles in the COMMERCIAL EVENT frame.

Fillmore gives another example [25] of possible frames that *live* can occur in:

1. Those lobsters are alive - the LIFE-DEATH frame
2. Her manner is very alive - the PERSONALITY frame
3. He gave a live performance - the ENTERTAINMENT-PERFORMANCE frame.

so that the appropriate understanding of *live naked girls* involves (3) not (1).

Langacker [26] uses the term domain in a sense very close to that of Fillmore's frame.

4.2 Mental spaces, frames and blends

The term mental space was coined by Gilles Fauconnier, and is described in *The Way We Think* [11]. A mental space is a transitory 'state of mind' which occurs when someone is thinking about something, and is a mental representation of that situation. Sometimes these are unique, but they often have elements in common with previously experienced spaces. For example walking into an unfamiliar room involves the familiar notions of floor, wall, window and so on, but the arrangement and occupants of that particular room may be unique to that space.

We can relate the idea of mental space to that of frame as described above, if we think of a frame as an entrenched mental space. That is to say, if situations are repeatedly encountered with mental spaces which are structurally similar, a frame can usefully be generated. For an individual this means they 'get used to' such situations. If in a community this happens for sufficient individuals in it, the entrenchment occurs for the community as well.

Fauconnier relates mental spaces to each other in 'conceptual integration networks'. This is an extension and generalisation of Lakoff's [27] characterisation of metaphor as a way of thinking of new ideas in terms of existing concepts. Fauconnier usually describes these networks as 'conceptual blends'. A blend results from 2 or more different mental spaces being brought together to produce a new distinct one - the output space. The output space is not like what a food blender produces - it is not a mushed-up version of the inputs. Instead it has a precise structure consisting of certain elements from the input spaces which constitutes an 'emergent structure' - something in some way different from the inputs.

An example of a blend is the Computer Desktop. One input space is the world of the office with files, folders and trash cans, and the other is the world of computer processes such as deleting or printing a file, executing a program and so on. When this becomes familiar, we 'live in the blend', meaning that we think of the elements of the situation in the blend, not in the input spaces they came from. For example, dragging a file icon to the trashcan is thought of as 'how you delete a file', rather than having to expand the blend to think of the icon as standing for the file, rather than being the file, and the dragging as how you delete it, rather than a metaphor for doing so. In fact the Desktop blend is more complex than this - we are ignoring the visual aspect, and also the fact that a 'computer file' is itself a blend between binary data and a paper-based file, and in turn a paper file is a blend of paper and ideas. This exemplifies the typical situation where blends are made of blends.

4.3 Compressions

A compression is the result of a process which takes several mental spaces which are in some sense 'the same' and yields a new one. This is one of the mechanisms by which these conceptual integration networks can be creative and imaginative, and enable us to think in a way which would be otherwise impossible.

Fauconnier gives many examples of compressions [11]. There are many associated with ideas of time. For example consider

The Sun rises in the morning

Time 'really' has a linear nature - although it is only linear in a metaphorical sense. We experience a sequence of points in time - Monday morning, midday, evening, night, Tuesday morning and so on. From these mental spaces we construct a compression of those mornings into a single idea, that of '*the* morning'. In that phrase, 'the morning' is singular - yet it does not refer to a single actual morning. In fact it is a way of referring to all mornings - but these are thought of (and spoken of) as a single item, namely a compression.

A second example, taken from a newspaper article about plans to reform mid-wifery services:

> Under the plans women will also be attended by the same two or three midwives throughout their pregnancy, with one of them delivering the baby.

Here 'the baby' is a compression, across the fictive mental spaces containing babies born in the future.

A third example is from a newspaper article reporting an experiment where strong magnetic impulses impaired arithmetic ability:

> The study, which finds that the right parietal lobe at the right/back of the brain is responsible for dyscalculia, potentially has implications for diagnosis and treatment through remedial teaching.

Here 'the right parietal lobe" is a compression, referring not to one part of one brain, but a fusion of the anatomical characteristics of all human brains.

It could be argued that a compression is the same as an abstraction. There are similarities, but the difference is that an abstraction is a logical process consciously undertaken, whereas a compression is a way of thinking about a set of things which the individual or the community adopts, but is not aware of. This corresponds to the distinction between a literary metaphor and metaphorical conceptualisation as described by Lackoff [27].

5 Frames, mental spaces and programming

This section applies the idea of frames to the process of thinking about programs.

A programming novice considering a short piece of programming code is obliged to think about two things. These are the program text, and what happens when the computer executes that text. These correspond to some extent to the common programming concepts of 'compile-time' and 'run-time'. These refer to two events, when the program is being compiled and when it is being executed. The distinction is relevant, for example, to memory usage. With static memory usage, such as when a conventional array is used, the size of the memory used is fixed when the program is written and compiled, for example by the programmer declaring an array with a fixed number of elements. However in the case of dynamic storage (such as a Vector in Java) the amount of memory used can vary at run-time, with elements added to the Vector structure as execution proceeds.

However frames are psychological constructs rather than the 'factual' notions of compile-time and run-time. What might be called the text frame is the thinking associated with the (high-level) text of the program, while the execution frame is thinking about the program being executed. These are like the COMMERCIAL EVENT frame of Fillmore, which had slots for BUYER, SELLER, GOODS and PAYMENT. What are the slots for these two frames?

The text frame includes the following slots

TEXT - the actual text of the program in high level language form.
CONVENTIONS - conventions associated with program code. These include indentation and capitalization - for example in Java the convention that classes and interfaces start with capital letters, and nothing else does.
SYNTAX - the syntactic rules associated with the language in use, such as what type of brackets are used, how statements are separated, whether the language is case-sensitive, how identifier scope is established and so on.

PURPOSE - what the author of the code intended it to achieve. For example the purpose might be to find the maximum of 5 inputted numbers.

The execution frame includes these slots

INPUT - what data values are input as the program runs. This, like the following, is time-dependent, or more precisely, execution-unit dependent. In other words it makes a difference which point during execution the data values are presented.
OUTPUT - what data values are output
VARIABLES - what value each variable or storage location will have as execution proceeds.
EFFECT - a characterisation of what the program 'does', in terms of a mapping between input and output. For example a program might output the minimum of 5 inputted numbers.

For a 'correct' program PURPOSE and EFFECT are identical, whereas a bug yields
an EFFECT which differs from the PURPOSE.

The cognitive difficulty of handling the two spaces depends on the structure of the code.

5.1 Simple sequence - a one-one mapping between spaces

A concrete example of this in pseudo-code would be

```
x=2
y=3
z=x+y
output z
```

Most students (even with no experience of programming) find this extremely easy to understand and predict what the program will do. Why? Because there is a one-to-one mapping between the text and the execution mental spaces:

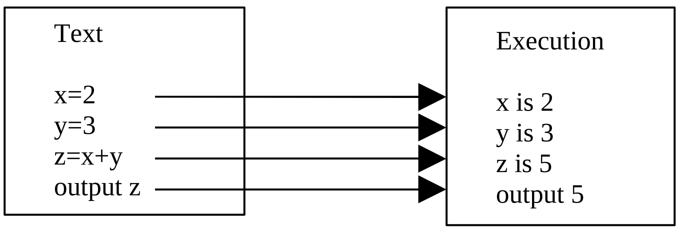

Fig. 1. The mapping between text and execution frames for a simple code sequence

This kind of program exhibits the idea of a high level language, namely the program statements in effect 'say' what the computer will do when they execute.

All the students in the group reported here found this program trivial. However in a separate study of students with low academic achievement levels on vocational courses, one student was found who said he was "muddled" by this. This is discussed later.

Regarding the program as a function mapping input to output, there is no input so the domain is the null set, and the output set has a single element, 5. So this is a constant function. Regarding a program as a function in this way is a common approach in undergraduate courses. However that is a formal model, to be compared with what is asserted here, which is a cognitive model. There is a recurring theme that the student must develop the appropriate cognitive model before the formal model makes sense.

5.2 Loops

If a program fragment contains a loop, there is no longer a one-to-one mapping between the two spaces. For example

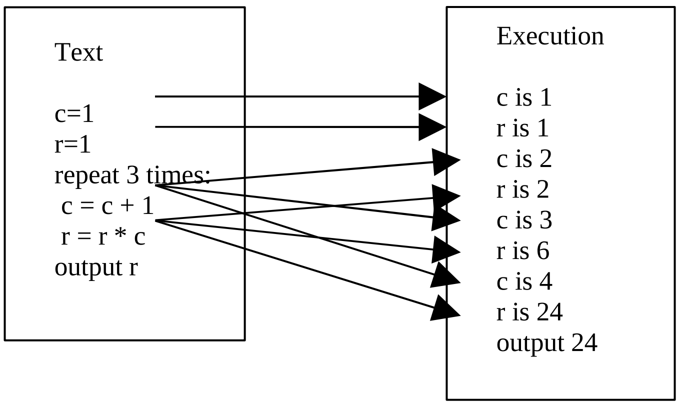

Fig. 2 The mapping between text and execution frame for a loop

In the interviews described below, the execution frame relates to the program trace which students were led through. The novice student can (and typically will) mechanically follow through the trace, and say the program outputs 24. However the question 'what does this program do' elicits from students who do not understand it no more than the answer 24, in that this is the calculated value when the loop ends.

Literal reality resides in the sequence of executed instructions in the execution frame. This has a subsequence which in term comprises of 3 sets of 2 instructions which are, to some extent, 'the same'. This is often referred to as the unrolled loop. These are *compressed* in the text frame to a loop of 2 instructions iterated 3 times - the rolled-up loop.

Which of course is obvious if you know about programming. However if you are a novice you only have the text frame, and must 'imagine' the execution frame. With experience of that imagining, the student can construct the idea that a loop is the compression of a set of instructions. With that idea, the student can reason about the statements in the loop in terms of what they will do. Without it, the student only has the rolled out statements in the execution frame, and no reasoning about them is possible, beyond simply what each individual statement does.

Explicitly, the student who knows that a loop is a compression can see that

```
 c = c + 1
```

increases c, not once, but every time - and then

```
 r = r * c
```

multiplies r every time by these increasing values of c. In turn they can see that the program calculates 1 X 2 X 3 X 4, rather than 24.

It is interesting to relate this to the idea of loop invariance, which is often used to show what programs with loops do. For example in this case the loop invariant would be that on the ith iteration, $r = i!$. This is true before the loop starts, and if it is true before an iteration, it is true afterwards. And it shows the fragment calculates n!. However this formal approach is not understood unless the student already has a cognitive grasp that a loop is a compression. This is an other example of a formal approach lying on top of an intuitive approach.

6 The data

Fourteen volunteers were interviewed in the autumn of 2007, in what was essentially a pilot study intended to provide a basis for identifying a suitable theoretical model. These were undergraduate students, with strong academic backgrounds, starting the first year of a degree in Computer Science. They were following a module which was an introduction to programming, including OOP and Java, and a parallel module concerned with data structures and algorithms. Seven of these students had little or no prior experience of programming, and these interviews took place towards start of the module, at a time when they had done virtually no programming in the course. Audio recordings of the interviews were made.

The interviews were semi-structured, starting with general questions about computers, programs and variables, and they were then presented with 5 short psuedo-code programs like those in 5.1 and 5.2, and the following:

```
x = 0
input n
while n is not equal to -99
          {
          if n > x then x = n
          input n
          }
output x
```

They were asked 'what would this program do?' If the student felt unable to answer this immediately, they were led through several traced runs, and then invited to summarise it.

6.1 Example

This example is chosen as being typical of the responses given. This subject has done no programming before. He starts by trying to deduce what the program would do - possibly, because he is silent most of the time. But he makes no progress (student statements are in italics):

> So what do you think that program would do?
> *Basically, because x was already set to zero, and then the while loop, n is bigger than zero, its going to be put here, .. er.. (15 second pause) I wouldn't know, I'm not sure (12 second pause)*
> You're not sure?
> *No it kind of gets me confused, if..*

So the interviewer introduces the idea of considering input values. The student needs a lot of help:

> OK OK if we.. one way to work this out is thinking what would happen if we put different numbers into it, so, x equals 0, input n, let's suppose we typed in 4, OK, so n would be 4,
> *Which is, err which is more than x*
> OK so it says 4 is greater than x, so
> *x is going to be 4,*
> OK so if we just jot that down, x is 4, yeah? Then input another value of n, so lets suppose we put in 6, OK? What will happen? It will loop around, and say n is not equal to -99, so we'll do it again,
> *Is it just going to print out 4?*
> We haven't got there yet. If n is greater than x, so 6 is greater than x, yes it is, x becomes 6. And we input another value. Now suppose we input 2, this time.
> *Still going to, x is going to be 2, its bigger than zero,*
> OK but x now is 6

Oh yes!
So it will say is 2 greater than 6, and its not
Its not
So that time it won't change x, so x will stay at that. Lets suppose we do it again and put in 3
Still 6
OK suppose we put in 7,
Its going to be print out 7
OK x becomes 7. Suppose we put in 1
7
Stays there, suppose we put next number -99
(5 second pause) Still 7
Yes - and what happens with the loop?
Yeah - its just going to print out the x,
It will come to here, yeah? so the loop will terminate there, OK? and we'll get 7, so it will output 7 in that situation.

The point here where the subject says 'Oh yes!' is where they are starting to see how x retains the previous highest value. But immediately after just this first run, the student has 'got it':

OK suppose we put in a different sequence and start again, at the beginning, and we put in 1, 2 , 3, 2 and -99. What output would we get?
3
3? Why would we get 3?
Because its increasing at first, and after 3, its smaller than the x value, so it will keep the 3, then its -99, and it will output the 3.
Yes OK. Um can we summarise this program then, can we summarise what this program does? You put some numbers in, what number do you get out?
The maximum number.

This student has understood after one trace run. Others needed up to 4 traces before they could summarise it correctly. But all followed the same pattern:

1. Carefully read the program text, and fail to understand
2. Follow through some traces
3. Become able to summarise it correctly

7 How does this perspective help?

The perspective of frames, blends and compressions is useful because:

1. It provides a framework for understanding the way novices try to comprehend programs, which comes from a general view of the way we think, not something which is specifically related to software. It would be unreasonable to suppose human cognition has characteristics concerned with programming which are different from general cognition.
2. It explains why some students cannot grasp even very simple programs as in 5.1. This is because they are only using the program text frame, and are not aware of the need to think about the execution frame.
3. It explains why some students are unsure about the difference between an 'if' and a 'while' statement. This seems to be paradoxical, since to us they are completely different. But it implies that some students do not know what a while loop is. This is possible, if they have not understood that a loop is a rolled-up version of repeated statements.
4. It explains why some students cannot construct simple code sequences involving loops, or reason about the effects of statements in given loops, if they have not achieved the implicit realisation that statements in loops are compressions.

8 Pedagogical implications

Some students find elementary programming very easy, and start to be able to use loops and conditionals in simple algorithms very quickly. These students grasp the idea of the dual frames of program text and the execution frame within minutes of their first programming class. But of course others find it incomprehensible. These suggestions are directed towards them.

1. Some students are not aware that what a program statement will do depends on the values of variables at that time - in other words they are not aware of the two frames of program text and execution. The fact that you have to take into account the values of variables is obvious to us, and so we do not explicitly say it.

A student can be helped to an awareness of the two frames by tracing the execution of program sections, either on paper or using a debugger. This helps the student to see, literally, the two frames.

2. Loops are typically introduced with examples such that the student must go through the following sequence:

program text with rolled up loop >> trace to unrolled loop >> compression of execution to loop statements >> understanding the loop

A more direct pedagogic path is

program text with unrolled loop >> compression >> loop

In other words presenting and stepping through programs with repeated statements, or getting students to write them, and then leading to the idea of compressing those multiple statements into a loop structure. This is adopting a constructivist approach, placing the student in a position where the idea of compressing repeated statements is reasonably obvious, and leading them to invent the notion of a loop themselves.

9 Developments

The interviews carried out during this pilot work are consistent with the central hypothesis, but there is not a lot of direct strong evidence to support it - not surprising in that the interview structure was established before the theoretical framework was selected. There is no part of any interview which one could point to and say - 'there, that proves that loop statements are seen as a compression'. In view of this the intention is to design a protocol which will elicit results more focused on the ideas of frames, blends and compressions with the intention of obtaining firmer evidence. It is also the intention to extend the scope beyond what is covered here to code structure in terms of functions and parameter passing, data structures such as arrays, and OOP ideas.

References

1. Biddle, R. & Tempero, E.: Java Pitfalls for Beginners SIGCSE Bulletin, Vol 30 No. 2. (1998).

2. Ragonis, N., & Ben-Ari, M. A. : A Long-Term Investigation of the Comprehension of OOP Concepts by Novices Computer Science Education Vol 15 No 3 September 2005

3. Eckerdal, A. & Thuné, M. : Novice Java Programmers' Conceptions of 'Object' and 'Class', and Variation Theory ITiCSE '05: Proceedings of the 10th Annual ITiCSE Conference, Monte de Caparica Portugal (2005).

4. Holland, S., Griffiths, R. & Woodman M. : Avoiding Object Misconceptions SIGCSE '97 : 28th Technical Symposium on Computer Science Education San Jose California (1997).

5. Fleury A. E. Programming in Java: Student-Constructed Rule SIGCSE 2000: 31st Technical

Symposium on Computer Science Education Austin Texas (2000).

6. Marton, F. & Booth, S.: Learning and Awareness. Lawrence Erlbaum, New Jersey (1997)

7. Booth, S.: Learning To Program - A phenomenographic perspective. PhD Thesis, Acta Universitatis Gothoburgensis 89:1992

8. Johnson, M.: The Body in the Mind: The Bodily Basis of Meaning, Imagination and Reason. Chicago University Press Chicago London (1987) xxii

9. Fleury A. E.: Encapsulation and Reuse as Viewed by Java Students. SIGCSE 2001

10. Douce, C. : Metaphors We Program By. Proceedings of the 16th Workshop of the Psychology of Programming Interest Group. Carlow, Ireland (2004)

11. Fauconnier, G. & Turner, M.: The Way We Think: Conceptual Blending and the Mind's Hidden Complexities. Basic Books, New York (2003)

12. Fauconnier, G. & Turner, M. : Conceptual Integration Networks. in *Cognitive Science*. Volume 22, number 2, pages 133-187. (1998).

13. Turner, M.: Compression and Representation, in Language and Literature (ed. Dancygier B.) vol. 15 no. pages 17 to 27 (2006)

14. Lakoff, G., & Nuñez, R. E.: . Where Mathematics Comes From. Basic Books, New York. (2000)

15. Nuñez, R. E. : Creating mathematical infinities: Metaphor, blending, and the beauty of transfinite cardinals, in Journal of Pragmatics (eds Coulson, S. & Oakley, T.) vol. 37 no. 10 (2005)

16. Alexander,J.C. Mathematical Blending , draft pdf
http://www.case.edu/artsci/math/alexander/pdf/alexander_blending_mathematics.pdf accessed July 2008

17. Imaz, M. & Benyon, D.. Designing with Blends: Conceptual Foundations of Human-Computer Interaction and Software Engineering. MIT Press. (2007)

18. Veale, T. & O'Donoghue, D. Computation and Blending, in Cognitive Linguistics (eds Coulson, S. & Oakley, T.) vol. 11 no. 3 pages 253-282 (2000)

19. Piaget, J.: Play, Dreams, and Imitation in Childhood. W.W. Norton. New York (1962)

20. Bartlett, S.F.: Remembering: A Study in Experimental and Social Psychology Cambridge. University Press Cambridge (1932)

21. Minsky, M.L.: A framework for representing knowledge. Massachusetts Institute of Technology A.I. Laboratory. (1974)

22. Abelson, R & Schank R., Scripts Plans Goals and Understanding: An Inquiry Into Human Knowledge Structures. Lawrence Erlbaum Associates Hillsdale NJ (1977)

23. R. S. Rist: Learning to Program: Schema Creation, Application, Application, and Evaluation. In Fincher S. & Petre M. (eds.): Computer Science Education Research. RoutledgeFalmer, London (2004)

24. Fillmore C.J.: Frame Semantics. In Linguistics in the Morning Calm. Hanshin Seoul,(1982) 111-137.

25. Fillmore, C.J.: Scenes-and-frames semantics. Linguistic Structures Processing, 59 (1977)

26. Langacker, R.: Foundations of Cognitive Grammar: Volume I: Theoretical Prerequisites, Stanford University Press, Stanford (1999)

27. Lakoff, G. & Johnson, M.: Metaphors We Live By, University of Chicago Press, Chicago (1980)