

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 validity of the framework in this application is discussed, and future work is outlined.

1 Introduction

This is concerned with how first year computer science undergraduates 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 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. Consequently an attempt is made to avoid a narrow focus on aspects such as features of Java contrasted with other languages, but instead to place the study within a general cognitive model of the development of sets of inter-related concepts.

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. This paper attempts to develop that.

OOP concepts are highly metaphorical and closely interlinked, making a psychological interpretation difficult. For a more straight-forward context, the work described here concerns students' responses to a short piece of pseudo-code which does not involve OOP ideas. This was used as a way of investigating how students approached the question of what a computer program 'did'. It was found possible to interpret their responses using the framework of conceptual integration networks.

2 Nonliteral Meaning

When a student is learning OOP and Java, they are primarily engaged upon understanding the meaning of the ideas involved. Consequently there is a need to examine meaning. Johnson [6] 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 is an excellent example of Objectivist, propositional 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 Java 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 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 [7] where a student is describing his reactions to a program altered so that the data members are public and there are no accessor methods:

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 refer to what it says. This is particularly the case with OOP, which is entirely metaphorical. However ordinary discourse about computing is also figurative, conventionalised so deeply as to make it difficult to recognise. For example, computers do not (literally) 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.

But far more simply than this - a loop is not literally a loop.

3 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 idea.

3.1 Frames

The idea of a frame is related to the notion of a schema, a term which has been used by several psychologists to denote variations on the theme of a structured mental representation. Piaget [8] uses the term scheme for patterns of activity in infants in what he calls the sensori-motor stage, and Bartlett [9] demonstrated the role of schemata in memory and recall.

Minsky [10] 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.

Related to this is the idea of a script by Schank and Abelson [11], 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 [12] 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 [13], 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 [14] 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 [15] uses the term domain in a sense very close to that of Fillmore's frame.

3.2 Mental spaces, frames and blends

The term mental space was coined by Gilles Fauconnier, and is described in *The Way We Think* [16]. 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 [17] 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 mashed-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.

3.3 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 [16]. There are many associated with ideas of time. 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 (for example) into a single idea, that of 'the morning'. This enables us to say things like 'The Sun rises in 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.

Having constructed those compressions, we can link them together to construct a cyclic notion of time which involves the compression we call 'the day'. This example is apposite, because there is a great similarity between the compressions of time and the statements in a loop, which are compressions of executed statements.

4 Frames, mental spaces and programming

This section makes some assertions about some cognitive aspects of programming from a novice's perspective. The evidence for and validity of these assertions is considered later.

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.

4.1 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
```

Almost all 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:

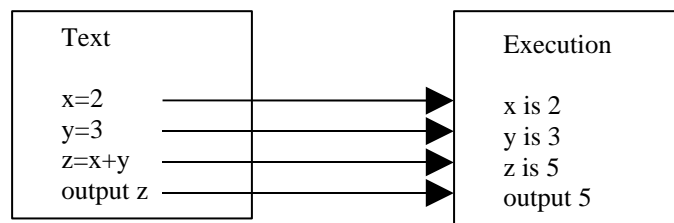


Fig. 1. The mapping between text and execution frames for a simple code sequence

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.

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.

4.2 Loops

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

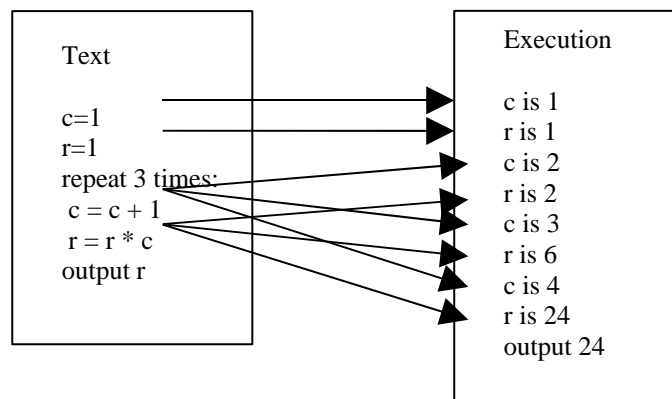


Fig. 2 The mapping between text and execution frame for a loop

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 no more than the answer 24.

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 $1 \times 2 \times 3 \times 4$, 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 i th 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.

5 The data

Fourteen volunteers were interviewed in the autumn of 2007. These were undergraduate students 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.

They were presented with this pseudo-code:

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

Two example interview transcripts are presented here - the rest were similar.

5.1 First example

The interviewer usually started by reading through the code with the subject like this example (interviewer statements are in normal text, and student responses are in italics):

There are 2 variables in this program, one called x and the other n. So we have x = 0, input n, that means read in a value for n from the keyboard, and that would be a number, while n is not equal to -99, we have a loop, so we do from there to there in this loop, so long as n is not equal to -99, and we have an if, if n is greater than x, n equals x, then we've got another input n, so here we are invited to type in another value, and the loop continues, until n is -99, and at the end of it we output x.

Yes

Then we have the key question:

So what would that program do?

Err, it could, in terms of what it would do if you put a value in for n? How it would work through, or what it would do in terms of what it would do here in the middle between these curly brackets?

Either

So this is an example of the question not having a clear meaning to this student, who had done no programming before. He suggests 'what would it do?' might refer to the execution of the code steps, but at this point it is very vague.

Nevertheless he spontaneously has the idea of a 'trace', involving imagining actual input values being entered and following what would happen :

Well we could put a value for n in, provided n is not equal to -99, it would run this loop, and for example if n was 5, 5 is greater than x, so x becomes 5, and then you input a new value for

n, say 6 to go with a corresponding, a consecutive value, so 6 would go in here, 6 is greater than x, so x becomes 6, this would be a never-ending loop, because it would keep going up and up, but if you were working with negative numbers it would stop, because you would get to, oh no you wouldn't, because if you put -1 in, -1 is not greater than 0, so it would output x,
OK

Well that's how I looked at it.

Here he is trying to deal with the loop. Because he inappropriately fixes on a positive sequence starting at 5, he reasons the loop will not end. But then he entertains the idea of inputting a negative number to stop it, but he is side-tracked by the realization that then the conditional would behave differently, and x would not be changed, and seems to link this with the loop ending. However the question 'what does this program do?' still has no clear meaning. The interviewer provides some guidance which leads the student to the idea of input being independently chosen by thinking of an analogy with a game :

OK now we can put in any numbers that we want to,

Yeah

Yeah, and they don't need to be in a sequence.

OK. I suppose the program there, could be to do with, it could randomly, the user could randomly select a number, or there could be something that generates numbers according to what the user of that program does, and then those values would run through

OK yeah

And then depending upon what the user was doing it could be in a game etc and then the moves of the user would determine the values of n, and then it could be that the output was something for the computer to do.

The interviewer then offers some input data to consider:

Now suppose we ran this, and I put in, the numbers 7 8 4 2 -99. Does it stop?

Yeah..

I'm putting in 5 numbers, it stops does it?

I would say so

Why?

Because when the 7 goes in at the end, 7 is greater than 0, then input the next number, which is 8, x becomes, sorry x becomes 7, x takes the value of n, 8 is greater than x, so x takes the value of 8, then n takes the value of 4, 4 is not greater than 8, so x does not take the value of n, but you could input another value of n, because there is no else condition, so it would stop when n was -99, and then x would be outputted.

So the student has traced this through to completion, but needs to work through it again (silently) to see the actual outputted value:

OK OK what then is the value of x? The outputted value? If I typed in these, what would come out?

Err.. (9 second pause) x would be 8

8?

I think, 7, yeah 8.

We try another input set:

OK now suppose I ran the program again and I put in, err, 12 3 13 1 -99. What would be the output for those numbers?

(16 second pause) The output would be 13.

13?

That's if I'm carrying out the program correctly?

Yes, you are correct. So what does this program do?

The student has still not yet reduced this completely. The looping is clarified but the output is not related in general to the input data:

The program carries out the operation according to the variable n, and then if it becomes equal to -99 at any stage, then it will output the value of x,

So the student is now clear about this ends when -99 is input, but all he can say is that x is output (as the code states), and cannot say what x is. We do a third run. The grasp is still not firm:

OK suppose we do it another time, and we put in 3 22 -99, what will I get out?

22

22?

Looking at it, I'm not sure whether the loop will just keep running, because even when n becomes -99, that could just stop - no, no ignore that actually

When we put in the -99, it then says, while its not equal to -99 do this, if it is equal to -99 we stop

It goes onto the next line

OK so we've been through this program three times, can you see some kind of pattern, here we get 8, here we get 13, here we get 22.. so which number do we get?

-99 in all of them

As being the last value which is put in?

That's when the if condition effectively finishes

So still the focus is on ending at -99, not the output. Finally we have it:

OK suppose we put in 1 8 23 97 31 48 12 -99 what number gets to be output?

97

Why?

Because its the largest number. It would increase normally here, but then when n is 97, its greater than x which is the previous value, which is 23, then none of the others are greater than that.

So if we summarise this, what does this program do, in summary what does this program do?

It selects the largest value, it will always output the largest value.

5.2 Second example

This subject also 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:

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 the consideration of inputting -99 needs more thought.

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.

6 Validity

At the heart of this is the idea that a novice programmer is obliged to deal with the two frames of program text and computer execution, and that the relationship between these in the context of a loop involves the notion of compression. Is this true?

Nothing as absolute as that is claimed. In the Popperian tradition proof is impossible, and the best we can do is to show that a theory is wrong by demonstrating it yields a prediction experimentally shown to be false. In this case the assertion is that the ideas related to conceptual integration networks are a useful framework for interpreting the development of students' ideas, and that interview transcripts are consistent with that framework.

In particular, this viewpoint explains why a weak student fails to understand loop constructs. Loops are nearly always presented with an emphasis on syntax, and with the loop as it is written in program code. The student has to unroll the loop into its iterations, then construct the compression which takes these back to the loop statements as written in the program text. Only then can the student grasp what a program with a loop will do.

7 Implications and Developments

These students go through the following sequence:

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

Some students have no problem with this, but some find it difficult to grasp the idea of a loop. A more direct pedagogic path is

program text with unrolled loop >> compression >> loop syntax

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.

With experience of dealing with code fragments, many students develop the ability to see what they do directly, without rehearsing a set of dry run traces. We can view this as a consequence of the development of a blend, putting together effective aspects of the input spaces of program text and execution frame - with the implication that experiences of dealing with code fragments enable the student to 'live in the blend' and make deductions in that blend as to what the fragments will do. This needs more work on students at this stage.

Further on, the target is OOP as implemented in Java. It is hoped that notions of class, object, inheritance and polymorphism, and misconceptions associated with these, can be interpreted in this framework.

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. Johnson, M.: The Body in the Mind: The Bodily Basis of Meaning, Imagination and Reason. Chicago University Press Chicago London (1987) xxii
7. Fleury A. E.: Encapsulation and Reuse as Viewed by Java Students. SIGCSE 2001
8. Piaget, J.: Play, Dreams, and Imitation in Childhood. W.W. Norton. New York (1962)
9. Bartlett, S.F.: Remembering: A Study in Experimental and Social Psychology Cambridge. University Press Cambridge (1932)
10. Minsky, M.L.: A framework for representing knowledge. Massachusetts Institute of Technology A.I. Laboratory. (1974)
- 11.

