Complexity is in the Brain of the Beholder: A Psychological Perspective on Software Engineering's Ultimate Challenge

Iyad Zayour, PhD
Faculty of Science, Lebanese University, Lebanon
Email: iyad.zayour@ul.edu.lb

Imad Moukadem, PhD
Computer Science, Gulf University for Science &Technology, Kuwait
Email: moukadem.i@gust.edu.kw

Issam Moghrabi, PhD
MIS, Gulf University for Science &Technology, Kuwait
Email: moughrabi.i@gust.edu.kw

"Beauty in things exists merely in the mind which contemplates them."

David Hume, 1742

***Abstract*—Complexity of software has been largely studied as a property of the code. We argue instead that complexity is a psychological phenomenon and should be studied from this perspective. The psychological literature however is structured in a way making of little practical usefulness. We propose a model based on isolated psychological facts connected by intuitive reasoning to fight complexity in a practical way. In this model, complexity corresponds to occurrences of cognitive overload in the working memory (WM), the bottleneck of cognition. Reducing complexity can be achieved by relieving the WM of some load by explicitly representing the internal mental constructs using external media such as software tools. We present a case study in which we used this model to produce a tool to reduce the complexity in program comprehension for large software systems. The tool was used in an industrial setting. We present here the mental constructs targeted and the details of the tool.**

***Index Terms*—Software psychology, Complexity measures, Program comprehension, Reverse engineering**

I. Introduction

Working with software is difficult, mainly because software is inherently complex [1]. Complexity is a central issue in software engineering, a *raison d'être*, one can argue. After all, if programs were simple, would ever such a discipline exist? In the software literature, complexity is discussed mainly in terms of code metrics. It is considered as a property of the code itself, ignoring that it is actually a human psychological phenomenon. It is the human intellect (of the beholder) that is the subject of the experience of complexity. Yet the discussion of complexity in software as a psychological phenomenon has been largely ignored.

The feeling of complexity or difficulty is not specific to working with software; it is a part of any problem solving process, a topic that is well studied in many disciplines especially in psychology. Apparently, the discipline of psychology should have a lot to offer to win the battle against complexity. Yet, it seems that very little practical psychological knowledge has been passed to the software engineering community. Perhaps there is a conception that dealing with complexity from the psychological perspective should involve crossing disciplines. The study of human intellect is the realm of psychology, while, as software engineers, we are more educated to work with software artifacts. The idea of crossing into a new discipline can be intimidating; we are more comfortable with the segregation of disciplines.

Psychology, in particular, can be very inhospitable for outsiders; it is a science that is based on rigorous experiments based on validating hypotheses in labs or lab-like settings [2]. Simple comprehensive theory or rules of thumb that can provide practical help for outsiders, like us the engineers, are not so common.

II. Psychology for Software: The Gap

Obviously, there is a gap between what psychology provides in form of rigorously validated conclusions and the kind of intuitive knowledge needed to guide us in our fight with complexity. Bridging this gap could be a valuable achievement; psychology is a science with over 100 years of accumulated knowledge that is supposed to help in dealing with a problem that is primarily psychological.

In bridging this gap, some vacuum has to be filled with conclusions that may be less scientific than what is the norm in psychology. However, there should be no fear of being less scientific: new trends in the philosophy of science encourage valuing theories based on their problem solving efficacy rather than on their scientific rigor [3]. Moreover, the alternative to relaxing the scientific rigor, as we propose, is the total absence of any guidance from scientific facts as we already have when we develop tools for helping with complexity. For example, the area of reverse engineering has seen little success in achieving its objectives. Many scholars argue that this disappointing result has been largely due to the fact that reverse engineering tool developer's base their design on their intuitive assumptions of what can be useful in reducing complexity in the absence of any formal guideline or criteria for evaluating success [4].

We argue that, in order to arrive at a theory or model that is practical enough to aid in software, connecting accurate dots (rigorous facts of psychology) with not-so-accurate lines (intuitive assumptions) is more likely to reveal a picture that is more accurate than a drawing without any demarcation points (the dots).

In this article, we try to present an intuitive model for fighting complexity that is based on psychological evidence and yet practical enough to influence or even guide software engineers in their fight with complexity. We try to connect the set of fragmented lab-proven facts, which can be individually of little practical value, into one comprehensive practical theory.

We start by giving an overview of some psychological facts that are instrumental to understand the nature of complexity (the dots) as seen by the psychology field. We then build on these facts (connect the dots) to propose a model of understanding and fighting complexity in terms that can be mapped into software design decisions. Finally, we describe how we applied this model/approach in a case study.

III. The Dots: Psychological Facts About Complexity

To find facts useful in dealing with complexity, the human cognitive system responsible for information processing is the place to start. Unfortunately, the logical anatomy of the human cognitive system is complex and subject to many theorizations about its functional components and their interactions.

Within the large psychological body of literature describing cognition, one logical component of this system has been shown to be so central in understanding the psychological aspects of complexity – the working memory (WM). The WM has very interesting characteristics that makes it intimately related to our interest in complexity:

- It is the mental workbench that is the place of all conscious activities and problem solving [11].
- It is well-characterized as a place of limited and scarce resources. It is widely considered to be the bottleneck for the human cognitive system [5].

WM itself has been subject to different theorizations (see related work section), yet some facts have accumulated significant empirical support about resource utilization and performance in WM:

1. The number of information elements or concepts that can be stored and used in problem solving in WM is very limited [14].
2. The more information is stored, the worse the overall performance gets [6].
3. The longer duration the information are stored (beyond 20-30 seconds when active rehearsal becomes needed), the worse performance gets [12].

As these facts show, the performance of the WM, and consequently the whole human cognitive performance, depends critically on minute details such as small differences in the number of items it stores and on the duration they are stored.

A. Connecting the Dots

To connect the above dots, we propose that much of our perception of complexity is directly related to the load on the WM resources. As such, a problem becomes complex when its memory representation requires enough resources that approach or exceed the WM capacity. In particular, it is the requirement to *simultaneously* store elements in the WM that tests its capacity. How many elements exactly and for how long is not important from our point of view as long as these elements and their storage duration are seriously limited compared to the magnitude of problems we need to solve in software engineering.

Accordingly, we propose that managing complexity depends on maintaining an adequate level of cognitive load (CL) and in particular avoiding cognitive overloads. Yet the question remains about the possibility of managing CL, can we choose to avoid overloads?

The answer seems simple when we depend on intuition: we do that every day. When our WM capacity is overloaded we resort to external artifacts. For example, when we need to store a piece of information that is large beyond our WM capacity (e.g. a 9-digit telephone number or a shopping list), we write it on a paper. The paper thus becomes an extension of the WM that extends its storage capacity and takes over some of its storage load. The paper can be considered as a cognitively inexpensive and resource abundant node within one distributed cognitive system to which storage load can be delegated. This delegation also happen in processing tasks such as when we use a hand held calculator to delegate some of the CL of arithmetic so we can free our cognitive resources for higher order task like calculating a financial statement. In addition to papers and calculators, a spreadsheet or even software are all examples of external artifacts to which we can delegate CL if they can properly integrate with our mental processes to form one cohesive distributed cognitive system.

B. CL and Complexity

To illustrate how the load on memory translates to the psychological notion of difficulty or complexity, consider the exercise of adding two numbers mentally. Start with adding a 2-digit number to a 1-digit number, say 77 +8. This can be easily performed mentally without any help as the elements of the problem representation fit within the WM capacity. However, when we increase the number of digits to four, say (77+89), the task starts to be perceived as difficult for some and even intractable for others. As the number of elements that need to be simultaneously retained in the WM increases (beyond 4 or 5 digits), most people find such problem to be difficult or complex.

So how do we manage complexity in such situations? Again, by using a pen and a paper, the many-digits problem becomes tractable. The paper relieves the WM from the load incurred in retaining the digits in memory since the digits visibility on paper substitute for their mental visibility. There will be no more a need for all the digits to be retained *simultaneously* in the WM for enough time until the mental calculation finishes.

The level of success in delegating or integrating external artifacts depends, however, on the ability of these artifacts to faithfully represent the implicit internal cognitive constructs that we build in our WM. The eye's view of these constructs should be very similar to what the mind would visualize. In fact, some recent research in brain neuroscience showed a striking similarity in brain activities between in-memory visualization and optical vision [7]. Mentally visualizing an object was found to produce a very similar brain signature to the signature produced by actually looking at the same object. Taking this into consideration, the term *externalization* becomes more appropriate as the goal becomes to externalize imagery from internal memory to external artifacts.

IV. An Approach for Handling Software Complexity

So how can we apply this theory/approach on our software problems? Can we sub-contract loads from the WM in a way that makes the complex problems that we face in software become less complex? Unfortunately, the nice parallelism in our example between mental and paper addition is not expected to be easily found in software. Identifying the mental constructs used during software tasks that can be externalized may not trivial. Moreover, faithfully representing these constructs can also be challenging. In the next section, we describe an approach and a case study in which we applied our approach in one of the most complex tasks of software engineering.

A. Managing CL

Perhaps one of the best places where complexity is intensely prominent is the area of program comprehension during software maintenance of large legacy systems. This is an inefficient process mainly because of the difficulties in comprehending the code of the software systems. Trying to understand someone else's code can be highly taxing on the WM due to the unfamiliarity factor since unfamiliarity prevent capitalizing on the information in the long term memory (chunking) to create a smaller footprint in the WM (another fact from psychology).

The classical way to address the inefficiency in software maintenance has been to develop reverse engineering tools that attempt to extract relevant knowledge from source code and present it in a way that breaks the code complexity and thus facilitates comprehension. However, these tools have a major "low adoption" problem among software engineers (SEs) in industry; developing tools that makes program comprehension easier has proved to be not so easy task. The failure to produce effective tools is, in our opinion, due to the absence of understanding of the true nature of complexity. There is a gap between the real source of difficulties a SE faces and what the reverse engineering tool developer intuitively assumes.

Program comprehension is predominantly a cognitive process, so instead of basing our tool design on intuitive assumptions about what make program comprehension easier; we present an approach that is based on what we are proposing in this article: reduce cognitive overloads that are the actual source of complexity by externalizing them to an external artefact – a reverse engineering tool.

B. The Approach

Our approach to break the complexity of program comprehension involves the following steps:

1. Characterize the mental processes used during program comprehension. Characterizing how programmers understand programs is a thread in computer science literature called "cognitive models of program comprehension" [8]. A cognitive model describes the major internal cognitive activities in a generic way. Knowledge provided by these models can be useful but, as we find out, they are often not detailed enough to identify problems that can be addressed by a tool. Therefore, we did our own work by analyzing the work practices of SE during software maintenance, focusing on how SE's comprehend programs from a cognitive perspective.
2. Identify situations and tasks where cognitive overloads occur. Such situations are identifiable by simple techniques such as external observation of SE work practices, by asking the SE to identify such situations, or even by introspection.
3. Analyze these tasks trying to identify the nature of the mental structures causing the overloads. The goal is to identify, among other things, the implicit processing constructs and operations that go on in the WM.
4. Design a tool that explicitly represent the mental constructs of the overload-causing tasks and externalize from the WM whatever possible sub-activities it can.

V. THE CASE STUDY

We applied our approach in a telecommunication company in order to help it reduce the cost of maintaining one of its large legacy systems. Minor maintenance requests take weeks to finish since most of the time is spent on comprehending the system. We analyzed the work practices of maintainers looking for sources of inefficiency.

We identified one of the situations of high inefficiency when small corrective maintenance tasks are assigned to entry level SE's who have little familiarity with code or understanding of the structure of the system.

The SE's were typically given a description of maintenance request in domain language. Their initial goal was to localize the code related to the maintenance request so they can modify it. This mostly requires mentally executing or comprehending the code (bottom up) to determine if the code is related to the maintenance request or at least leading to the related code.

We noted that the mental execution involves the tracing of the control flow at the level of routines more than at individual lines. The SE's follow the routine call hierarchy at varied levels of depth. They drill down when a routine is not clear or when it seems relevant; otherwise they move forward at high level in the call hierarchy.

In analyzing why this activity was perceived as so difficult and why it caused frequent overloads, we noted the following cognitive sources of difficulties:

1. Storage overload: the SE's, while trying to mentally explore the execution path of software as part of understanding its behavior, need to keep a mental map of who calls who— a kind of call tree. The map needs to be presented mentally in totality since understanding the meaning of a function depends on what functions called it and what functions it calls. This calling tree aggregates together to one concept representing the functionality to be achieved. Every part (routine name and relative position) of the call tree contributes to the overall grasping of the code executed. The necessity to keep all these elements in memory *simultaneously*, particularly when the depth of the call tree is significant, made this activity cognitively overloading.
2. High retention cost: the comprehension of code that executes in a certain scenario requires following the logical relation created by the execution (calling) flow between routines that are stored all over the software system (in different files and directories). The linear order of actual code execution has to be mentally reconstructed from delocalized static code. This requires finding each one of the delocalized piece of code (e.g. the calling routine), retaining it in the WM, and then finding the next related piece of code (the called routine) so that an overall map of the relationships can be mentally constructed. In moving from acquired information to find the next related one, significant time may elapse particularly since primitive search tools were used. This retention incurs significant CL and in many cases, as we observed, the retained information (e.g. calling routine) may fade from memory before the related information is reached thus breaking the whole comprehension process that requires that all the related pieces of information to be *simultaneously* present in the WM.

Interestingly, a similar finding (see related work) has been identified in the learning literature. The *Split Attention Effect* theory [9] states that the use of physically separated information sources that cannot be perceived simultaneously causes a higher cognitive load on WM due to the need to mentally integrate the information.

VI. THE TOOL

Now that the mental constructs that overload the WM have been identified, we need to externalize it to be represented by an external medium—a reverse engineering tool. That is, if SE's, during comprehension, mentally construct call trees then let's delegate this load to the tool and let the tool do that instead, so the scarce resources of the WM can be freed for higher order mental tasks.

Call trees have often been used in trying to understand program behavior (e.g. call stacks and traces), but unfortunately not in cognitive-friendly forms. For example, a call tree can be constructed by a step-wise debugger that permits following the call flow; however this has been shown to be highly disorienting and not useful for program comprehension tasks [10]. A call tree that shows in a panoramic way all the call relations during a scenario is an attractive representation as it explicitly represents the code in its order of execution, thus removing the WM load caused by code delocalizing.

The tool that we developed takes a call trace (a log of entries representing the called routine within a scenario) and processes the trace in order, generating a call tree (see figure 1). Processing was needed since the call traces, even for small executed scenarios are inherently large and are of little value in their raw forms.

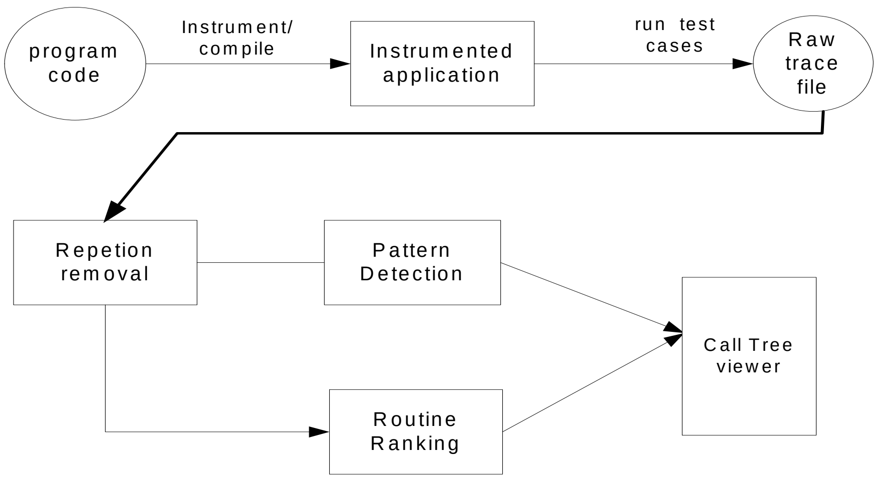


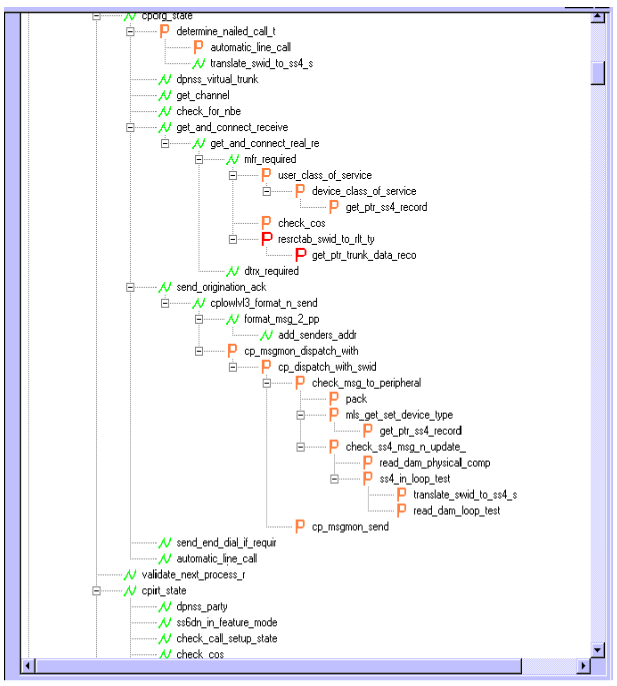
Figure 1: A diagram showing the various steps of trace processing

Some of the techniques to ensure this parallelism between the mental and tool representation included:

1. Compression: mentally, a human would not visualize 100 nodes for the same routine if it was called within a loop of 100 iterations. Thus, we removed redundant call entries caused by loops and recursion.
2. Selective level display: To parallel the selective exploration of call depth that we observed in SE’s, we allowed the user to control the depth of the call tree using the collapse/expand at each node (much like file explorer), so the user drills down in the call tree whenever they need to. The user can also choose a specific level of call depth to apply to the whole tree.
3. Selective routine display: Not all routines contribute to the comprehension equally, some routines are more important than others i.e. they yield more information in constructing the higher level concept the SEs are trying to assimilate. To permit the user to hide the less-important routines, we developed heuristics that approximate the human evaluation of importance of routines (ranking) and permitted the user to hide the least important on a continuous spectrum.
4. Learning: contiguous routine call sequences were found to recur in the traces. These sequences represent patterns that when comprehended at their first encounter should be abstracted, learned then remembered by the user. To extend this ability of the user to remember what he has learned, the tool permits the user to choose a sequence of call entries (a pattern) and replace it with a high level description. This description, will then replace all other occurrences of this sequence in the traces.

VII. Evaluation

The various techniques used in the tool did compress the call traces to a size that made them readable [11] and proportional to the functionality covered. However, the most relevant evaluation is related to perception of SE’s of how much difficulty has been reduced.



```
cporg_state
    P determine_nailed_call_t
        P automatic_line_call
          translate_swid_to_ss4_s
      dpnss_virtual_trunk
      get_channel
      check_for_nbe
      get_and_connect_receive
          get_and_connect_real_re
              mfr_required
                P user_class_of_service
                    P device_class_of_service
                        P get_ptr_ss4_record
                P check_cos
                P resrctab_swid_to_rlt_ty
                    P get_ptr_trunk_data_reco
              dtrx_required
      send_origination_ack
          cplowlvl3_format_n_send
              format_msg_2_pp
                  add_senders_addr
            P cp_msgmon_dispatch_with
                P cp_dispatch_with_swid
                    P check_msg_to_peripheral
                        P pack
                        P mls_get_set_device_type
                            P get_ptr_ss4_record
                        P check_ss4_msg_n_update_
                            P read_dam_physical_comp
                            P ss4_in_loop_test
                                P translate_swid_to_ss4_s
                                P read_dam_loop_test
                    P cp_msgmon_send
      send_end_dial_if_requir
      automatic_line_call
  validate_next_process_r
  cpirt_state
      dpnss_party
      ss6dn_in_feature_mode
      check_call_setup_state
      check_cos
```

Figure 2: The call tree generated by the tool where P nodes corresponds to a pattern not yet abstracted

The SE’s appreciated the new ways to view the software provided by the tool that explicitly represented an invisible domain (the dynamic domain i.e. the execution of the software) that could be only be constructed mentally with difficulty. Some noted that although they had a mental conception of how the call tree looks like for some familiar code, they were still amazed of what the call tree actually looked like in the tool. This amazement seems to reflect that they were never able to mentally construct large-enough call trees; a plausible claim given that, in our subject system, a depth of more than eight levels in the call tree was not uncommon. One programmer described the tool as a panoramic debugger, since a stepwise debugger gives a small window of visibility on what is executing, while the higher order view, such that provided by the tool, has to be mentally constructed. Overall, the consensus was that the tool significantly facilitated the comprehension of the system. In particular, it facilitated the kind of comprehension that was especially difficult during software maintenance.

VIII. Background and Related Work

A. *The Evolution of WM*

The psychological understanding of the WM and the resources it holds has dramatically evolved from its view as a short term holder of a limited number of information elements. One of the most credible and popular models for WM is that suggested by Baddely [12], who views it to be more like a mental workbench.

Under Baddely's model, the WM is made out of a central executive and two slave subsystems: the phonological loop and the visuo-spatial sketchpad. The central executive is thought to be the primary workbench area of the system where mental work of all sorts is done. It initiates a variety of mental processes, such as decision making, retrieval of information from long-term memory, reasoning and language comprehension.

The phonological loop is a sound-based system that can hold and recycle small quantities of information; it corresponds to a short-term rehearsal buffer. The visuo-spatial sketchpad is a specialized slave system that holds visual or spatial codes for short periods of time.

Baddely considers that the central executive can be thought of as a pool of mental resources available for any of several different tasks but which is limited in overall quantity. Each of the two slave systems also has a limited pool of resources.

Resources are shared in one direction, from the central executive down to either the phonological loop or the visuo-spatial sketchpad. The central executive shares its resources with the slave systems when either one of the slave systems becomes overburdened (with an overly demanding task) and needs extra resources. However, when the central executive shares its resources, it often ends up having insufficient capacity to do its own work.

Recently Baddely [13] added to his model a fourth component, the episodic buffer, which holds representations that integrate phonological, visual, and spatial information, and possibly information not covered by the slave systems.

B. *Cognitive Load and Learning*

One of the areas that managed to do well in utilizing the psychological knowledge is learning and the design of the learning material. We can find in the domain of learning comprehensive theories with strong empirical evidences covering the relation between WM load, performance and the notion of complexity:

The theory of relational complexity of Halford [14] defines relational complexity as the number of independent elements or variables that must be simultaneously considered to solve a problem. Halford argue that relational complexity reflects the cognitive resources required to perform a task. The processing difficulty of any task is "the number of interacting variables (i.e., dimensions or arguments) that must be represented in parallel to perform the most complex process in the task". Halford was even more specific when he showed that only 4 concepts can be integrated in the WM of an adult *simultaneously* as part of the mental problem representation.

The well-known Cognitive Load theory (CLT) [15] suggests that learning happens when there are no cognitive overloads. They provide techniques for reducing WM load in order to facilitate learning. They also propose that users may get an information overload when there is too much information that is presented in parallel.

The split-attention effect theory [9] states that when learners have to split their attention between disparate sources of information, then these sources of information have to mentally integrate before the instructional material can be rendered intelligible. This process of mental integration is likely to impose a heavy cognitive load and thus impede learning.

IX. Conclusion

Complexity is primarily a psychological phenomenon. Psychology, a well-established science, should help much more than it is already doing. The problem is that the literature of psychology is not formulated in a way that can provide practical help in the software business as it is made in terms of isolated facts (dots) that fall short of being comprehensive theories of practical value.

In this article, we showed that, using some intuitive reasoning, these dots can be connected to draw a map that can be useful in guiding the software community in fighting its eternal enemy – complexity. The approach, case study and the reverse engineering tool presented a demonstration of how this can be accomplished.

While performing their intellectual tasks, humans create very fluid mental representations that are not always easy to replicate with more concrete media. Software, more than any other medium experienced by humanity before (namely paper), can also be very fluid and thus promising in paralleling memory representations.

Yet, lessons learned from reverse engineering, whose main goal is to break the complexity of existing software, can also be used in forward engineering. Complexity is not specific to software maintenance; it can be part of the entire software life cycle. We need to conceptualize systems beyond our WM capacity starting from the architecture, passing by analysis and design till development.

People intuitively use sketches and diagrams to visualize the big picture, or more formally they use modeling languages like UML. Sketching or modeling perhaps can be viewed as a way to break concentrations of complexity by creating external representations, like a pen and paper in mental addition. If that become well established, then the perspective we provided in this article may be useful in resolving the religious debate about the usefulness of modeling (agile vs. formal), answering questions such as how, when and how much modeling can be useful.

REFERENCES

[1] F. Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering”, *IEEE Computer*, vol. 20, 4 April (1987)

[2] W. Kintsch, *Comprehension: a Paradigm for Cognition*, Cambridge university press, p. 2. (1998)

[3] L. Laudan, *Beyond positivism and relativism*, Westview press, (1996)

[4] M.A. Storey et. al., “Cognitive design elements to support the construction of a mental model during software exploration”, *Journal of Systems & Software* , vol.44, no.3, Jan. (1999)

[5] Card S., Moran T., Newell A., *The Psychology of Human Computer Interaction*, Erlbaum Assoc., Hillsdale NJ, p. 392. (1983)

[6] M. Ippel, “Cognitive Task Load And Test Performance”, http://www.ijoa.org/imta96/paper52.html

[7] J. Decety, “Do imagined and executed actions share the same neural substrate?”, *Cognitive Brain Research*, 87-93 (1996)

[8] A. Von Mayrhauser, A.M. Vans ,“Program Understanding Behavior During Adaptation of Large Scale Software”, Proceedings. 6th *International Workshop on Program Comprehension* (1998)

[9] P. Chandler, & J. Sweller, “The split-attention effect as a factor in the design of instruction”. *British Journal of Educational Psychology*, 62, 233-246. (1992).

[10] B. Korel, J. Rilling , “Program slicing in understanding of legacy system”, ", *Proceedings Fifth International Workshop on Program Comprehension* (1998)

[11] I. Zayour, “Reverse Engineering: A Cognitive Approach, a Case Study and a Tool”, PhD thesis, p.117 (2002)

[12] A.D. Baddeley, Working Memory, Oxford: Clarendon Press (1986)

[13] A.D. Baddeley, “The episodic buffer: A new component of working memory?” *Trends in Cognitive Science,* 4, 417-423 (2000).

[14] S. Halford, H. Wilson, and W. Phillip, “Processing capacity defined by relational complexity: Implications for comparative, developmental and cognitive psychology.” *Behavioral Brain Sciences*, 21, 803-831. (1998).

[15] J. Sweller, "Cognitive load during problem solving: Effects on learning". *Cognitive Science,* 12 (2), 257–285(1988).