ESCAPE Meta Modeling in Software Engineering: When Premature Commitment is Useful in Representations

Jim Buckley[1], Chris Exton[1], Aaron Quigley[2] and Andrew LeGear[1]

[1] Department of Computer Science and Information Systems, University of Limerick, Limerick, Ireland
{jim.buckley, chris.exton, Andrew.legear} @ul.ie
[2] Department of Computer Science, UCD, Dublin, Ireland
{aaron.quigley@cs.ucd.ie}

Abstract. This paper introduces, and provides a cognitive basis for, a prototype meta-modeling process called ESCAPE. This process involves users Explicitly Stating their own model of an entity of interest, CAPturing an alternative or correct model of that entity and consequently re-Evaluating their own model. The paper shows the model's implicit, but already well established, use in the software engineering domain. In particular, it focuses on empirical work carried out in Software Understanding and Architectural Recovery of large commercial software systems using the Reflexion modeling process, which embodies ESCAPE meta-modeling principles. Finally, it suggests several areas where ESCAPE meta-modeling could be beneficially applied in software engineering.

1 Introduction

The process of developing software systems is information-intensive, as illustrated by the work of Nunamaker[22] and Teichroew[34]. In any development cycle, for example, one of the first activities is requirements analysis, where system developers must capture information about the system's domain and its functionality. This information can be obtained from tender documents or, more frequently, from potential system-users, through interviews, observation of work practices and system prototyping.

Later in the system's development, a large amount of the knowledge that the developers require can be obtained from the study of highly structured and, in many cases, highly complex system documents created or obtained earlier in the development cycle. For example, in 'White-Box' testing, the software testers must acquire a detailed knowledge of the control-flow structure of the source code, thus basing the test suite directly on the artifact to be tested [26]. Likewise, empirical studies of developers' software-maintenance behavior [29], [31] suggest that it is a very code-centric activity.

The importance of *searching* these structured documents, for information in a software engineering context, is highlighted by works like [10] [14], [39] [30] and [31]. Over several studies Singer and Lethbridge [31] found that searching 'implementation documents' (source code) was the most common activity carried out by developers in a tele-communications domain: In their report they showed that over 45% of all tool usage by experienced developers was in the form of system searches using grep.

In a similar vein, [39] found that developers spent approximately 35% of their time navigating between source-code dependencies. Indeed, they characterized software maintenance as a process of "collecting a group of task-relevant code fragments… navigating those code fragments… and repairing or creating the necessary code". Studies like these led Sim et al. [30] to characterize experts as task-oriented information seekers, performing searches when repairing code, reusing code, understanding systems, adding features and assessing the impact of changes.

1.1 Efforts to Facilitate these Information Requirements

The complex nature and size of the documents available to developers, as they develop and evolve software systems, places great demands on their information-seeking. Indeed, empirical evidence suggests that up to 90% of the entire development effort is spent simply reading, navigating and understanding the code [40].

Given the predominance of this activity, researchers in software engineering have recently begun to study programmers' information seeking in its own right. These studies have particularly focused on programmers involved in software maintenance, characterizing information seeking in terms of blocking factors, information sources, information requirements and information overload [12], [29], [23], [24]. However, these studies are still at a largely preliminary stage.

'Software Visualization' communities have sought to facilitate information-seeking by developing software visualization tools [1], [6], [17] [21], [28]. These tools typically generate various views and abstractions of software systems that the user can navigate around and between, based on the system's underlying structure.

This paper, while acknowledging the value of such tools, argues that, in some situations, *more* user interaction may be beneficial. It argues that, when dealing with experienced software developers, it may be beneficial for the visualization tool to require the user to explicitly state their expectations of the system before being presented with system representations. This argument is based upon the study of practices that have become productively embedded in in-vivo software engineering. In particular, the success of a technique called Reflexion modeling us used to illustrate the potential of this approach.

1.2 Paper Structure

This paper presents the proposed ESCAPE Meta-modeling process, a process derived from Software Engineering practice, where it has been shown to be successful. In section 2, we describe ESCAPE and in section 3 we show how the process is embedded in several key Software Engineering practices. Additionally, in section 3, we describe our empirical experiences using an ESCAPE-based Architectural Recovery approach in two commercial software development companies, and report on its successful adoption, by those companies. Finally, we suggest several novel applications for ESCAPE meta-modeling in section 4.

2 Escape Meta-modeling

Escape meta-modeling is based on creating a conflict between the expectations of the user with respect to some entity of interest and the actuality of that entity of interest. It is anticipated that such a conflict will have 'shock-value', prompting the user's curiosity and driving them to resolve the conflict, leading to learning.

As an illustrative example, consider holding out a pen in your hand at shoulder height and letting go. You would expect that the pen would drop to the ground. Imagine, instead, that the pen rises into the sky. Your model of how things should work has been shown to be incorrect, as it conflicts with the actuality of the situation. Curiosity is a natural response, as is the desire to reconcile your understanding of your world with the actuality.

The first step in ESCAPE meta-modeling (as shown in Figure 1) is to make the user Explicitly State their model of their entity of interest (the pen will drop). The second step is to CAPture information regarding the entity of interest (letting the pen go from a height). The third step then is for the user to Evaluate their expectation in the light of the captured information. If the 2 are aligned (the pen drops) then the user can have increased confidence that their model is correct. If the 2 don't align (the pen rises) the user may have to either re-assess his understanding of the situation (perhaps there is a large helium compartment in the pen) or he/she may have to change the underlying situation (for example moving outside of the gravity-free chamber). Either option finesses the user's understanding of their context.

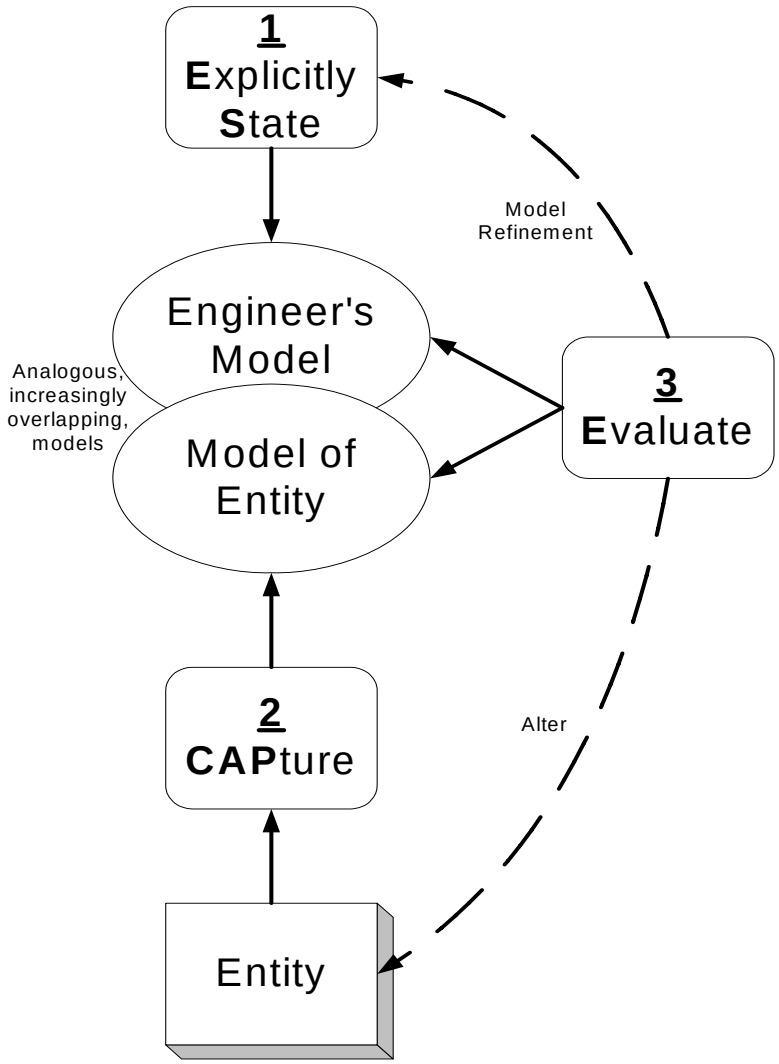

Fig 1. The ESCAPE Meta-Modeling Process

1.2 Psychological Basis

Unsurprisingly, there is a cognitive basis for this work. Piaget [36], for example, postulated that conflict is a driving force for learning and development. Cognitive conflict occurs when a particular endeavor requires more than can be achieved by

using existing knowledge or familiar strategies. Thus existing knowledge or abilities need to be amended or expanded. Likewise, Strike and Posner [37] contend that, in order for conceptual change to occur, the student must first be discontented with their current understanding (or mental model) as it no longer fits with what they observe. It is this dissatisfaction (or cognitive conflict) that motivates the student to consider alternative conceptual views that may result in a number of amendments or expansions to their understanding.

Consequently, cognitive conflict is a positive state, as it provides an impetus for us to either correct our own understanding or adapt the real world situation to comply with our own understanding in order to re-achieve a state of equilibrium. Given this reasoning, one surprising conclusion is that visualization tools might be more effective if, instead of presenting software visualizations to the end-user, they forced the end-user to state their expectations first and only then presented the visualization.

Other relevant cognitive theories include Vygotsky's 'zone of proximal development' [38] and Piaget's related notion of 'Dissonance' [42]. Vygotsky's 'zone of proximal development' refers to the difference between learners' current development level and the learners' potential level of development. Piaget hypothesized that if this gulf is to large, learning will suffer. So, returning to our earlier example of the falling pen, if the pen rises, an observer could hypothesize a number of possible reasons (the helium compartment, a fine string attached to the pen and to a roof beam) and further exploration may be prompted. However, if the pen turned into a nuclear warhead, the observer would probably be lost for any possible hypotheses and may just accept the phenomenum. This suggests that ESCAPE modeling is best employed by experienced users who could legitimately expect to have a model close to actuality. Such a situation would have lower dissonance, and the inconsistencies that did exist would be genuinely surprising.

However, fit-to-theory literature suggests that people tend to pay attention to evidence that confirms their thories and disregard evidence that contradicts their theories. As such, fit-to-theory suggests that ESCAPE modeling would be of lesser use. However, this can be (at least) partially addressed by using representations that emphasize where the inconsistencies exist.

ESCAPE also can be discussed in terms of Green's Cognitive Dimensions [7], in that it is an example of premature commitment – a dimension traditionally seen as affording lesser utility in representations. In this instance, premature commitment presents itself in the form of forcing users to explicitly state their model, before this model is (possibly) contradicted by the actual model. This contradiction may force the iterative reformulation of the user's initial model and the user may have to reformulate that model.

Consequently reformulation of the user's initial model should not be difficult (or 'viscous' [7]). In fact, ideally, it should be very lightweight, within the constraint that it allows users to fully express their model. Practice and empirical evidence from the

domain of software engineering suggests that, in this case, premature commitment is a positive attribute and this is further explored in section 3.

The viewpoint that people's models are initially inadequate is reflected in the quote attributed to George Box, the statistician: 'all models are wrong, some are useful' [3]. In most cases a model is considered good is if it allows us to structure our understanding in a useful manner. It is apparent, in many aspects of our everyday life, that most of us operate in the world using rather flawed or simplistic mental models. When these simplistic models are inadequate, they can cause difficulties. ESCAPE modeling is about facilitating the recognition that people's models are inadequate and rectifying that situation.

In a software engineering context, the consequences of the cognitive conflict in ESCAPE meta-modeling are principally dependent upon the types of disparity that lead to the discord. In some cases it can serve as the instigator of a change requirement that will lead to an amendment of some software artefacts, typically source code or perhaps a configuration element.

Conversely it may also result in a change of the Software Engineers' own model of how the system functions. In this case the task is not to realign the actual behaviour or structure of the software to the engineers mental model but to realign the software engineers own comprehension or mental model to that of the actual system. The scale and granularity at which this happens is closely related to the scale and granularity of the cognitive conflict.

3 Current application to Software Engineering

In this section a number of practices in software engineering, implicitly based on ESCAPE meta-modeling are presented. It should be noted that these exemplars are often considered core or best practice in software development and evolution, strengthening the suggestion that any underpinning model has a strong, if implicit validity.

- Prototyping: Here the requirements analyst Explicitly States the system that he thinks the users want. The users and the requirement analyst subsequently meet and the analyst CAPtures the divergences between his stated model and the user's. This forces a re-Evaluation of the analyst's model and may result in changes to the model or indeed changes to the user's expectations.

- In testing, best practice is for software engineers to Explicitly State their test-cases' input and output. The test cases are then run through the system, CAPturing divergences between the expected outputs and the actual outputs. The divergences force the tester to re-Evaluate the system in terms of its buggy output and possibly alter it. Alternatively, although less frequently, they may change the test suite to rectify errors in its formation.

- Paired programming. Although relatively novel, paired programming is quickly becoming a popular practice in software engineering [Lui and Chan 2006]. Here 2 or more programmers work together as a team when programming. Contextualizing this as an instance of ESCAPE meta-modeling, one of the two programmers (programmer 1) Explicitly States his/her model in discussions. The other programmer (programmer 2) provides the CAPtured model (not necessarily a correct model in this instance – just a different interpretation.) Programmer 1's model is altered in the light of programmer 2's model and criticism. Likewise, programmer 2's model is also refined in the light of programmer 1's comments. While it is possible that this just focuses the programmers on a joint, flawed model, the increasing popularity of the technique, suggests that the aggregate model represents an improvement on each individual's model.

The techniques reviewed to date, while illustrative of ESCAPE meta-modeling, could hardly be considered visualization techniques in their own right. However, Reflexion, as proposed by Murphy et al. [19] is a technique that adheres to ESCAPE in a visualization-tool context.

3.1 Reflexion Modeling

Software Reflexion modeling is a diagram-based, structural summarisation technique. It is supported by an Eclipse [5] plug-in called the jRMTool [9]. The technique is primarily aimed at allowing software engineers to gain a greater understanding of their software system's architecture. Software Reflexion modeling follows a six step process, illustrated in Figure 2:

1. The software engineer who wishes to gain a greater understanding of the software system hypothesises a *high-level* conceptual model of its structure. This model need not, in any way, reflect the current explicit structure of the system.
2. The computer extracts, using a program analysis tool, a dependency graph of the subject system's source code called the *source model*.
3. The programmer then creates a map which maps the elements of the source model onto individual nodes of the high-level model.
4. The computer then assesses the call relationships and data accesses in the source code to generate its own high-level model (called the Reflexion model). This model shows the relationships between the source code elements mapped to different nodes in the programmer's high-level model. This allows comparisons between the computer's model and the programmer's model and the tool can report this comparison in three ways:
    - A dashed edge in the Reflexion model represents dependencies between elements of the programmer's high-level model that exist in the

the source model, but were not placed in the programmer's high-level model.

- A dotted edge in the Reflexion model represents a hypothesized dependency edge of the programmer's high-level model that does not actually exist in the source model.
- A solid edge in the Reflexion model represents a hypothesized edge of the programmer's high-level model that was validated by the source model.

5. By targeting and studying the inconsistencies highlighted by the Reflexion model the programmer can either alter their hypothesized map, the high-level model or indeed the underlying system to produce a recovered model of higher consistency.
6. The previous two steps are repeated until the software engineer is satisfied that the recovered model is consistent with their high-level model.

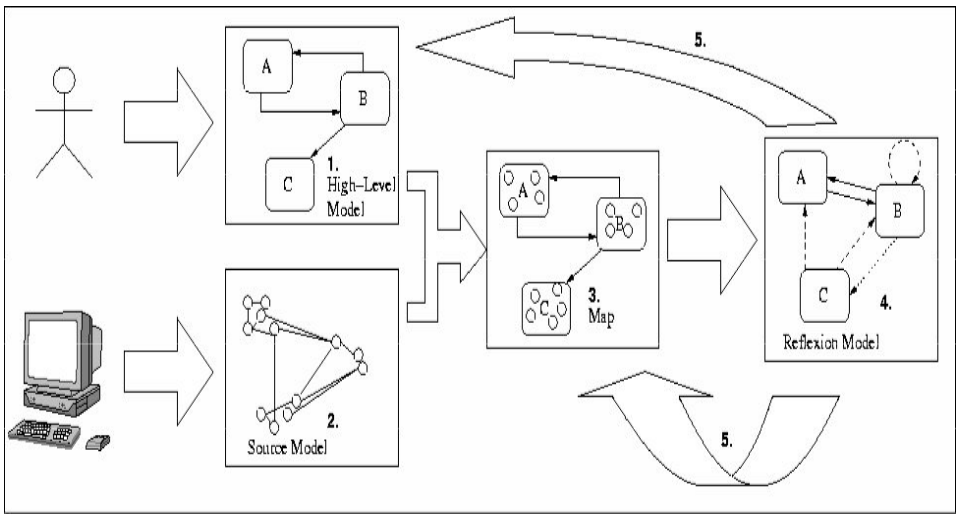

Figure 2 –The Software Reflexion Modeling Process

This software understanding method closely conforms to the ESCAPE meta-modeling framework. Specifically, it prompts the user to state their architectural model of the system, and reflects the actuality of the system back to them in terms of their model. Inconsistencies prompt the user's curiosity and thus drive them to re-evaluate and reconcile the 2 models.

Reflexion modeling is accompanied by promising results in facilitating the understanding of large software systems [19,20,13,15]. For example, in two experiments, detailed by Koschke and Simon [13], users are described as gaining an encompassing understanding of 100KLOC and 500KLOC compilers in 6 and 8 hours respectively. In Murphy and Notkin[19] study, a software engineer stated that, using Reflexion modeling, he gained an understanding of Excel in one month that would normally have taken 2 years.

3.2 Case Studies

This section presents empirical data derived from a number of case studies performed over a 1.5 year period in 2 Irish-based software development companies. One is a medium sized enterprise situated in the west of Ireland employing 35 people and specializing in Management-Process support software (henceforth referred to as company 1). The other company is a large multi-national situated in Dublin, employing over 300 software engineers on a wide range of software projects (company 2).

In total 3 case studies were performed, each involving an experienced software engineer from the companies, working on one of the company's large proprietary software systems, and using Reflexion modeling to carry out one of their assigned work-tasks with respect to that system. Details of the 3 sessions, obtained from a questionnaire before the study and from the study itself, are given in the table below.

Case Study	A	B	C
Company	1	2	2
Participant – Commercial Experience	5.0 years	2.0 years	2.0 years
Participant – Experience with the System	0.25 years	1.5 years	1.5 years
Participant – Domain Experience	6.0 years	1.5 years	1.5 years
Task	Achieving a better arch. understanding for maintenance	Achieving a better arch. understanding	Isolating the GUI
System – Domain	Warehouse Mgt.	Learning	Learning
System – Size	250 KLOC	500 KLOC	500 KLOC
System – Language	Progress	Java	Java
Session Duration (approx)	2.45 hours	2.5 hours	3.5 hours
Session Location	In-situ	In-situ	In-situ

Table 1: Case Study Profiles

Participants were given a short (20 minute) introduction to Reflexion modeling, where they used the technique on a small software system. To enable the technique, the jRMTool Reflexion modeling plug-in [19] for Eclipse was utilized. Both companies used Eclipse as their standard development environment and the jRMTool plug-in served to limit the artificiality of the situation. The plug-in provides automated abilities for creating and viewing high-level models, for mapping software elements to high-level model elements and for displaying the resultant Reflexion models. It also displays summary information regarding the edges of the model and unmapped software elements.

After this introduction, the participants were asked to undertake their scheduled work-tasks using the Reflexion modeling approach and were asked to state everything that came into their minds, as it came into their minds. Part of this data is presented later in this section to demonstrate ESCAPE modeling in practice. The sessions were sometimes broken for activities like coffee, lunch, interruptions by work colleagues and by higher-priority work that arose.

The 2 selections of quotes below are talk-aloud data, taken from the 3 software engineers as they performed their respective work-tasks. The first selection shows the users' positive impressions, and how they valued Reflexion modeling.

> *"I did try this same job about two months ago and gave up after two weeks."*
>
> *"It does such a good job of helping you understand the architecture of the system."*
>
> *"A really good tool for getting a high-level idea of a big amount of source code."*
>
> *"… great for spotting where dependencies were nonetheless ..."*

Indeed, in both companies, the practitioners acted as champions for the technique, prompting wider adoption among their colleagues. For example, in company B, 16 software engineers subsequently used this approach. It seems that Reflexion modeling was effective at reducing programmers' information seeking effort by allowing them easily model their system in a task-appropriate manner and focusing their attention at places where their model of the system is insufficient or inconsistent with reality.

The following quotes illustrate several times when cognitive conflict explicitly arose, when using the approach:

> *"Looks like there's a few links all right, from rest of system back to view, ...I was hoping there would be none, so that's a shame… lets look"*
>
> *"Pretty much on the mark about the one way direction. There's only four going back into the RF_SCREENS, curious to know what they would be [checks edge information]"*
>
> *"We've got 19 calls between XXX_util and 'rest of XXX,' which is interesting, right, we need to figure that out because . . . we'd need to find out what those 19 things are going back here, because they theoretically should belong in there [pointing to 'XXX_util'] as well."*

While they are illustrative, the quotes presented here are by no means exhaustive. Many more such quotations are available, on request from the 1st or 4th author.

3.3 Potential Utility in Software Engineering

There seems to be several other potential applications of this meta-model in software engineering. Specifically:

- Data Mining Social Networks: Large software companies often have organizational charts that detail the professional relationships between their personnel and between their teams. This could be considered an Explicit Statement of the expected (professional) social network. If some means of tracking the interactions of employees could b agreed (RFID cards, observation) then the actuality of their social network could be CAPtured. By comparing this model to the organizational charts, unexpected relationships between employees, and their teams, could be uncovered. Follow-up in depth qualitative analysis could lead to a greater understanding of the implicit relationships and dependencies between different business areas and between different individual personnel. Likewise, the lack of relationships between individual team members could be highlighted as an area of concern.

- Reflexion modeling is currently a static analysis technique. A dynamic analysis alternative would be for the QA department to instrument the system before they run through their test cases and to thus identify the source code executed for each test case. Hence, they could easily derive the source code executed for each desired function of the system. Later, when other programmers debug or evolve a specific function of the system, the CVS could track where they made changes. This explicit statement of the 'code that needs to be changed' could be compared to the code sets derived from the QA Department's test cases and divergences could be used to suggest the location of hidden ripple effects. While this work is similar in nature to [Wilde et al '95]'s Reconnaissance work and [Koschke 2004] Concept Lattice representations, it is novel in that both these techniques present their information to the user without forcing the user to state his / her assumptions up front.

- Open Source Project Management Analysis: Management style may change when a new team takes over an Open Source development from another. However, these changes may be implicit to the new management. Here the models compared are the work practices of the previous management and the work practices of the new management. While neither of the 2 models can be viewed as correct, in this example, divergences could be identified leading to re-evaluation of management styles.

- Gaming for System-Structural Knowledge: Games like Stellar Empires and Risk involve players trying for world / galaxy domination by attacking their opponents positions on a world map. However, instead of playing for world domination on a world map, you could play for system domination on a sys-

tem map. In this adaptation, the world map would be a call graph of your system, or perhaps a graph representing its calls, its inheritance hierarchies and its friendship relationships. In this case, the players' goal is to take over all methods/classes (the individual map locations) of the system. The essential idea here is that in moving their forces round the system, competitive advantage is obtained by knowing where your opponents can attack from. Thus, those players with the best structural knowledge of the system will win.

In an ESCAPE context, the programmers make an Explicit Statement of (their knowledge of) system structure, by moving their forces to (seemingly) defensible positions. Opponents may then shock the player by attacking through relationships that the player didn't anticipate. Such moves show a partial structural knowledge not in evidence in the player's original mental model of the system and force the player to re-Evaluate his/her model of the system. Games may be played by software team members or against a computer player, over long periods of time, resulting in prestige for the winner and increased structural knowledge for the team.

4 Conclusions

This paper has proposed ESCAPE meta-modeling as an effective framework for visualization in a software engineering context. It has showed where the framework is implicitly embedded in software engineering practice and software visualization, suggesting its effectiveness. Finally, it has shown several other possible application areas where the framework may provide valuable insights in software engineering.

References

1. Antoniol G., Fiutem R., Lutteri G., Tonella P., Zanfei S. and Merlo E. (1997). "Program Understanding and Maintenance with the CANTO Environment" Proceedings of the International Conference on Software Maintenance. pp 72-83.
2. Basili V. (1996). "The Role of Experimentation: Past, Present and Future". International Conference of Software Engineering. keynote speech.
3. Box, G.E.P., Robustness in the strategy of scientific model building, in Robustness in Statistics, R.L. Launer and G.N. Wilkinson, Editors. 1979, Academic Press: New York.
4. De Lucia A., Fasolino A.R. and Munro M. (1996). "Understanding Function Behaviors through Program Slicing". Proceedings of the 4th International Workshop on Program Comprehension. Eds: Cimitile and Muller. IEEE Computer Society Press. pp 9-19.
5. Eclipse IDE Homepage. http://www.eclipse.com [September 2006].

6. Gallagher K.B. (2000). "Working Session: Tools for Program Comprehension: Building a Comprehender's Workbench". Open Discussion during the 8th International Workshop on Program Comprehension. June 10-11, Limerick Ireland.
7. Green, T. R. G. (1989). Cognitive dimensions of notations. In People and Computers V, A Sutcliffe and L Macaulay (Ed.) Cambridge University Press: Cambridge., pp. 443-460.
8. Harrison W. and Basili V. (1996). "Editorial". Empirical Software Engineering. Vol 1, no 1. pp 5-11.
9. jRMTool Reflexion Modelling eclipse plug-in. http://www.cs.ubc.ca/murphy/jRMTool/doc/ [December 2003].
10. Kay J and Richard T. (1995). "Studying Long Term System Use". Communications of the ACM. Vol 38, no. 7. pp 61-68.
11. Kemerer C.F. and Slaughter S. (1997). "Methodologies for Performing Empirical Studies: Report from the Internation Workshop on Empirical Studies of Software Maintenance". Empirical Software Engineering. Vol 2. No. 2 pp 109-119.
12. Ko A. DeLine R. and Venolia G.. Information Needs in Co-located Software Development Teams. To appear ICSE 2007
13. Koschke R, Simon D. (2003) Hierarchical Reflexion Models. Working Conference on Reverse Engineering.
14. Koschke (2004) A Concept Analysis Primer
15. Le Gear A, Buckley J. (2005) Reengineering Towards Components with "Reconnexion." ESEC/FSE Doctoral Symposium.
16. Linos P., Helleboid Y., Lejeune P. and Tulula P. (1992). "A Software Tool for Understanding and Reengineering C Programs" 5th ACM SIGSOFT Symposium on Software Development Environments.
17. Linos P.K. and Courtois V. (1994). "A Tool for Understanding Object Oriented Program Dependencies". Proceedings of the 3rd Workshop on Program Comprehension. pp 20-29.
18. Lui K.M., Chan K.C.C.. (2006) Paired Programming Productivity: Novice-Novice Versus Expert-Expert. International Journal of Human Computer Studies. Vol. 64. no. 9. pp 915-925
19. Murphy G.C., Notkin D, Sullivan K. (1995) Software Reflexion Models: bridging the gap between source and high-level models. Symposium on the Foundations of Software Engineering pages 18-28
20. Murphy G.C.,Notkin D. (1997) Reengineering with Reflexion Models: a case study IEEE Computer 2(17) pages 29-36.
21. Ning, J.Q., Engberts, A. and Kozaczynski,W. (1994). "Automated Support for Legacy Code Understanding." Communications of the ACM. Vol 37, no. 5 pp. 50-57 .
22. Nunamaker J.F. (1971). "A Methodology for the Design and Optimization of Information Processing Systems". AFIPS Conference Proceedings. Vol 38 pp 283-293.
23. O'Brien M.P. and Buckley J. (2005) "Modeling the Information-Seeking Behavior of Programmers – An Empirical Approach" In Proceedings of the 13th International Workshop on Program Comprehension. pp 125-135
24. O'Shea P. and Exton C. (2005). "The Role of Source Code within Program Summaries describing Maintenance Activities", In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 05, Brighton UK
25. Phfleeger S.L. (1998). "Software Engineering Theory and Practice". Prentice Hall.
26. Pressman R.S. (2000). "Software Engineering, A Pratctioner's Approach" Fifth Edition. McGraw-Hill.
27. Rajlich V. (1994). "Program Reading and Comprehension". Proceedings of Summer School on Engineering of Existing Software". Bari, Italy pp. 161-178.

28. Sajaniemi J. (2000). "Program Comprehension through Multiple Simultaneous Views: A Session with VinEd" Proceedings of the 8th International Workshop on Program Comprehension 2000. pp 99-108.
29. Seamen C. B. (1999). "Qualitative Methods in Empirical Studies of Software Engineering". IEEE Transactions of Software Engineering. Vol: 25, no 4. pp 557-572.
30. Sim S.E., Clarke C.L.A. , and Holt R.C. (1998). "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers." in Proceedings of the 6th International Workshop on Program Comprehension. pp 180-187.
31. Singer J., Lethbridge T., Vinson N and Anquetil N. (1997). "An Examination of Software Engineering Work Practices." In Proceedings of CASCON '97 pp 209-223
32. Sommerville I. (2006). "Software Engineering (7th Edition)". Addison Wesley.
33. Strike, K., and Posner, G. (1992) A Conceptual Change View of Learning and Understanding. In L. West & R. Hamilton (Eds.), Cognitive structure and conceptual change (pp. 211-232). London: Academic Press
34. Teichroew D. (1974). "Problem Statement Analysis: Requirements for the Problem Statement Analyzer". Chapter from Couger J.D. and Knapp R.W.. "System Analysis Techniques".
35. N. Wilde and M. C. Scully (1995) Software Reconnaissance: Mapping Program Features to Code. Journal of Software Maintenance: Research and Practice, 7(1):49–62.
36. Piaget, J. (1985). The Equilibration of Cognitive Structures: The Central Problem of Intellectual Development. Chicago: University of Chicago Press.
37. Strike, K., & Posner, G. (1985). A conceptual change view of learning and understanding. In L. West & R. Hamilton (Eds.), Cognitive structure and conceptual change (pp. 211-232). London: Academic Press.
38. Vygotsky L.S. (1978). Mind and Society: The development of higher psychological processes. Cambridge MA: Havard University Press.
39. Ko A.J., Aung H.H., and Myers B.A. (2005) Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. ICSE 2005.
40. Erlikh L. (2000). Leveraging system dollars for E-business. IT Professional. May/June 2000. pp 17-23.