Software Comprehension – A Review & Research Direction

Michael P. O'Brien

Department of Computer Science & Information Systems
University of Limerick
Ireland

E-mail: michaelp.obrien@ul.ie

Technical Report UL-CSIS-03-3

November 2003

Abstract

Comprehending computer programs is one of the core software engineering activities. Software comprehension is required when a programmer maintains, reuses, migrates, reengineers, or enhances software systems. Due to this, a large amount of research has been carried out, in an attempt to guide and support software engineers in this process.

Several cognitive models of program comprehension have been suggested, which attempt to explain how a software engineer goes about the process of understanding code. However, research has suggested that there is no one 'all encompassing' cognitive model that can explain the behavior of 'all' programmers, and that it is more likely that programmers, depending on the particular problem, will swap between models (Letovsky, 1986).

This paper identifies the key components of program comprehension models, and attempts to evaluate currently accepted models in this framework. It also highlights the commonalities, conflicts, and gaps between models, and presents possibilities for future research, based on its findings.

1. Introduction

Software comprehension is "the process of taking computer source code and understanding it" (Deimel & Naveda, 1990). Burd et al (2000) defines it as "the

activity of understanding existing software systems". Muller (1994) defines software comprehension as "the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself, to models of the underlying application domain, for maintenance, evolution, and reengineering purposes". However, the author enhances this definition, defining software comprehension, as "a process whereby a software practitioner understands a software artefact using both knowledge of the *domain* and/or semantic and syntactic knowledge, to build a mental model of its relation to the situation".

Current research into software comprehension models, suggests that programmers attempt to understand code using the somewhat clichéd taxonomy of 'bottom-up comprehension', 'top-down comprehension', and various combinations of these two processes.

Bottom-up comprehension models, propose that as source code is read, abstract concepts are formed by chunking together low-level information (Pennington, 1987), (Detienne, 2002), (Shneiderman & Mayer, 1979). In other words, understanding is built from the bottom up, by reading the code and then mentally chunking or grouping these lines of code into higher-level abstractions. The bottom-up model of software comprehension primarily addresses situations where the programmer is unfamiliar with the domain. Several 'top-down' models of software comprehension have been proposed to address the alternative situation, where the programmer has had some previous domain exposure. Essentially, these top-down models of comprehension suggest that the programmer utilises knowledge about the domain to build a set of expectations that are mapped on to the source code (Brooks, 1983), (Shaft, 1992), (Good, 1999).

It is unlikely, however, that programmers rely on either one of these strategies exclusively. Instead, the literature suggests, they subconsciously adopt one of these to be their pre-dominant strategy, based on their knowledge of the domain under study (von Mayrhauser & Vans, 1997), (von Mayrhauser et al., 1997), (Shaft & Vessey, 1995), and switch between comprehension processes as cues become available to them (von Mayrhauser and Vans, 1995). In fact, Letovsky refers to programmers as 'opportunistic processors' to reflect the ease with this they change their comprehension strategies in response to external cues and stimuli (Letovsky, 1986).

Several theories have been developed to aid in program understanding. Essentially, this paper attempts to summarise and evaluate a selection of the most acceptable theories of program comprehension and highlight any commonalities, conflicts, or gaps between them. It discusses these strategies in detail, explaining, Pennington's program and situation models, Shneiderman and Mayer's syntactic and semantic understanding model, Soloway & Ehrlich's approach, Detienne's work, Letovsky's theory of opportunistic comprehension, and Brooks' domain mapping model. Von Mayrhauser & Vans' integrated meta-model of program comprehension is also discussed.

2. Software Comprehension Models

Although software comprehension models differ significantly in their emphasis, they all consist of four common elements, namely, a knowledge base, a mental model, external representation, and some form of assimilation process (see Figure 1).

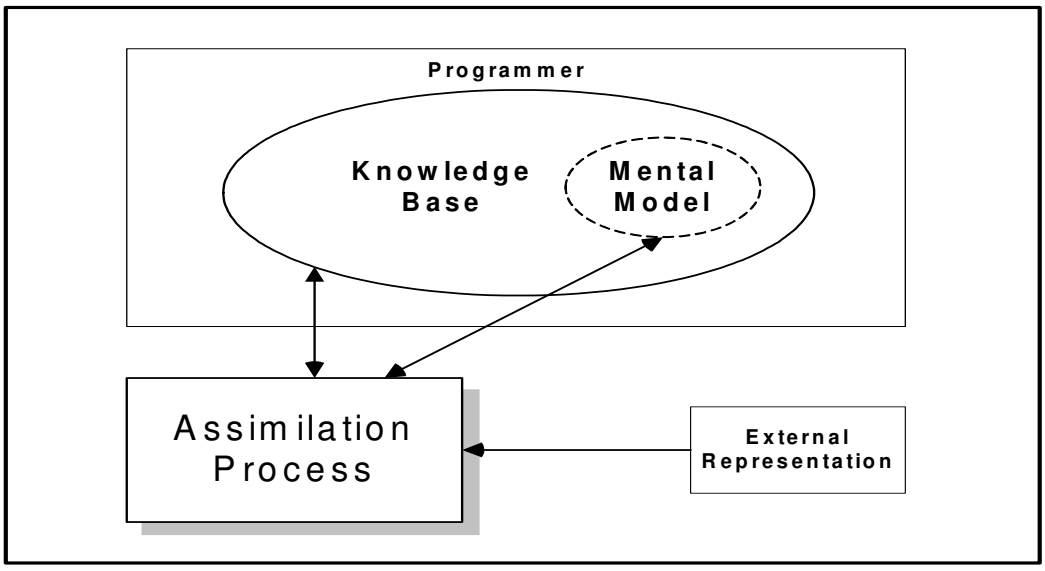

Figure 1 - Components of Software Comprehension Models

Firstly, external representations are essentially, any 'external' views available in assisting the programmer when comprehending code. This external support may be in the form of system documentation, the source code itself, expert advice from other programmers familiar with the problem domain, or indeed, any other source code similar to the code under observation.

Essentially, the 'knowledge base' (see section 2.1) can be defined as the programmer's accumulated knowledge before they attempt to understand the code. It may consist of an understanding of the domain, general information that may be pertinent to that domain, along with programming standards and practices. The knowledge base develops and expands as the level of programmer understanding deepens.

The programmer's existing or current representation (therefore understanding) of the system under study is referred to as their 'mental model'. Using the knowledge base, mental model, and external representations, the assimilation process continuously updates and augments the programmer's mental model. The 'assimilation process' is the actual strategy, which the programmer employs to comprehend the source code (Davis, 1993). One method of assimilation is where programmers' hypotheses are refined and elaborated during comprehension (Brooks, 1983).

Effectively, hypotheses can be defined as 'ideas or explanations for something that is based on known facts but has not yet been proved' (Cambridge English Dictionary). Brooks (1983) suggests, that programmers use an iterative, hierarchical method of generating, refining, or repudiating hypotheses, at different levels of abstraction, during code comprehension. Letovsky (1986) subsequently investigated the role of hypotheses in code comprehension, referring to hypotheses as 'conjectures'. Conjectures are associated with a level of certainty, ranging from a complete guess, to complete certainty. This level of certainty affects the way in which they are revised or discarded from the mental model at a later stage during the comprehension process.

2.1 Knowledge Base

Many researchers have attempted to understand the knowledge, programmers' use, and require, during program comprehension. Knowledge can be obtained at any abstraction level, and can thus range from low-level implementation details to higher-level structural or semantic information. Much research has been carried out to investigate the nature of the programmer's knowledge base and has shown that the knowledge base consists of; domain expertise (Brooks, 1983); coding knowledge –

beacons (Brooks, 1983), (Rist, 1986) and plans (Wiedenbeck, 1986); syntactic & semantic knowledge (Schneiderman & Mayer, 1979), (Gellenbeck & Cook, 1991), (Soloway, 1984); domain goals (Von Mayrhauser & Vans, 1997); rules of programming discourse (Soloway, 1984); and schemas (Letovsky, 1986), (Detienne, 2002). These knowledge types span many different [individual] theories/models of software comprehension (see Figure 2).

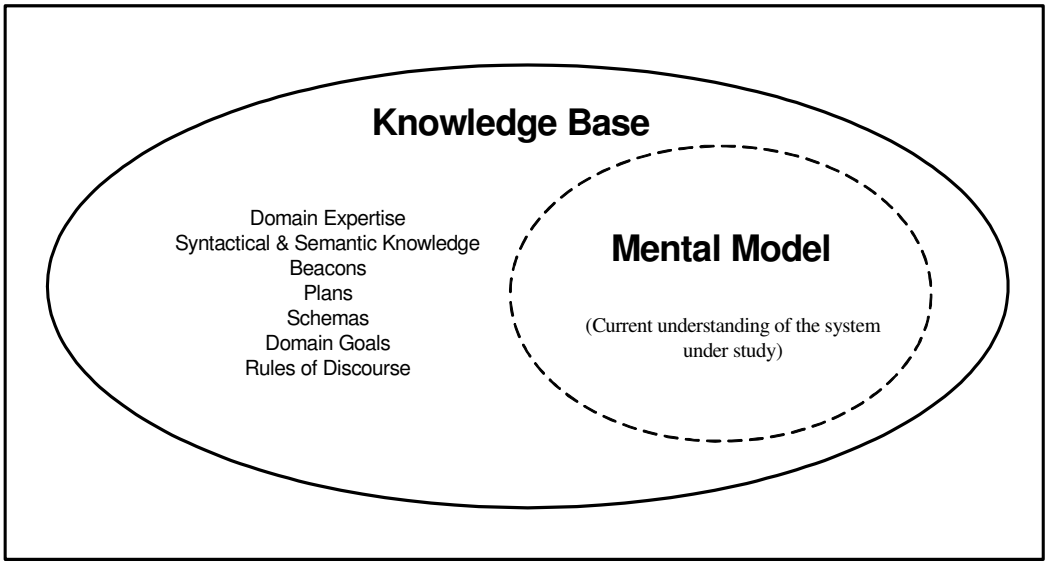

Figure 2 – The Knowledge Base (Expanded from Figure 1)

2.1.1 Domain Knowledge

Brooks (1983) proposed that programmers use 'domain knowledge' to reconstruct knowledge about the domain of the program and map this to the actual source code itself. This reconstruction process is theorized to be 'top-down' and hypothesis driven, where the programmer is familiar with the domain and is thus able to pre-generate hypotheses (expectations), which may come from other programmers, their own expertise, or the design documentation (external representations). Essentially, Brooks' proposes that programmers construct mappings from the task domain (application domain), through one or more intermediate domains, to the programming domain. Although Robson et al (1991) claim this process is a 'bottom-up approach, working from the source code using beacons to form an initial hypothesis', Brooks theory asserts that an initially vague and general hypothesis (Figure 3, step 1), often generated from the program name, alone, is refined and

elaborated upon based on information extracted from the program text and other relevant documentation. This primary hypothesis then produces a cascade of subsidiary hypotheses (Figure 3, steps 2 & 3), with the decision of which hypothesis to pursue, based on the programmer's motivation for comprehending the program. Cascading continues until it produces a hypothesis that is specific enough that the programmer can match (verify) it against the source code (Figure 3, step 4). Specifically, the programmer begins the verification of a hypothesis when the hypothesis deals with operations that can be associated with visible details found in the source code.

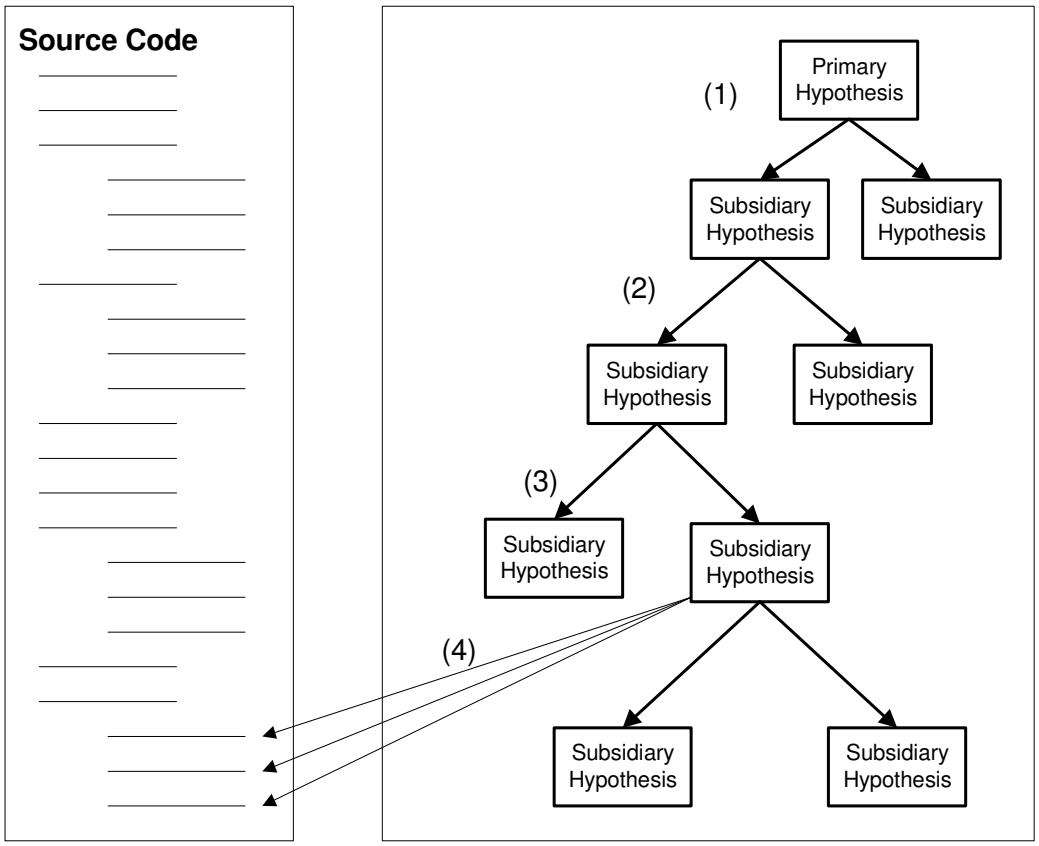

Figure 3 – Brooks' Hypotheses Generation & Verification Process

2.2.2 Beacons

Beacons, first identified by Brooks (1983), but empirically explored by Weidenbeck (1986), are essentially, recognisable or familiar features within the source code, which act as cues to the presence of certain structures or plans. As the programmer gets more experienced, they become more aware of these segments of

code and, during code study, recognition of these segments causes the programmer to suspect the presence of a specific plan. Programming plans can be described as "program fragments that represent stereotypic action sequences in programming to achieve a specific goal" (Rist, 1986). (A good example is a clichéd sort routine in the code). The programmer may then search other parts of the code to validate the 'plan' suggested by the beacon. Brooks (1983) states that beacons can themselves lead to validation of hypotheses and to generation of subsidiary hypotheses or more rapid verification of later hypotheses.

2.2.3 Syntactic & Semantic Knowledge

Shneiderman & Mayer (1979) suggest an overall cognitive framework for describing behaviours involved in program composition, comprehension, debugging, modification, and the acquisition of new programming concepts, skills, and knowledge. An 'information processing model' is suggested, which includes a long-term store of syntactic and semantic knowledge, and a working memory in which problem solutions are constructed. Syntactic knowledge is language dependent and concerns the statements and basic units in a program. Semantic knowledge is language independent and is built in progressive layers, initially using syntactic knowledge & the code, until a mental model is formed which describes the purpose of the system. The final mental model is acquired through the chunking and aggregation of other semantic components and syntactic fragments of text.

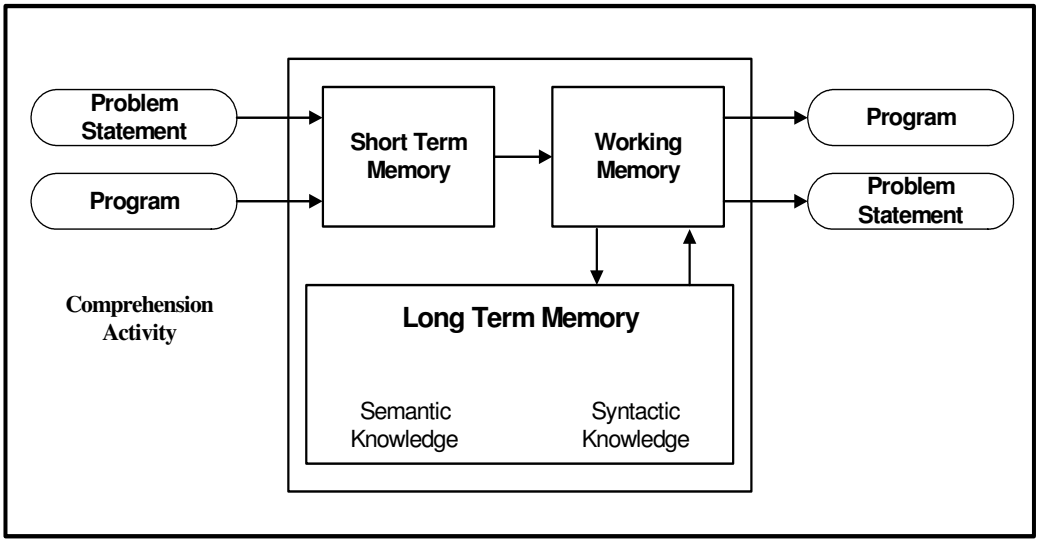

Figure 3 – Syntactic & Semantic Knowledge

According to this theory, during comprehension, the programmer takes program statements into short-term memory (see Figure 3). Syntactic knowledge is then brought from long-term memory to develop a low-level understanding of these code constructs. Semantic knowledge is then brought from long-term memory to match the syntactic constructs and identify their function and is built-up by chunking to form higher-level semantic units. In other words, the semantic knowledge associated with individual statements is chunked together to form higher-level semantic units. This chunking process eventually leads to an overall understanding of the program goals.

2.2.4 Plans & Rules of Programming Discourse

Soloway & Ehrlich (1984) suggests that 'expert' programmers possess two types of programming knowledge: *programming plans*, as mentioned above (Weidenbeck, 1986), (Gellenbeck & Cook, 1991), (O'Brien & Buckley, 2001) and *rules of programming discourse*. Soloway & Ehrlich (1984) suggest that program comprehension is primarily based on opportunist recognition of programming plans (see Figure 4).

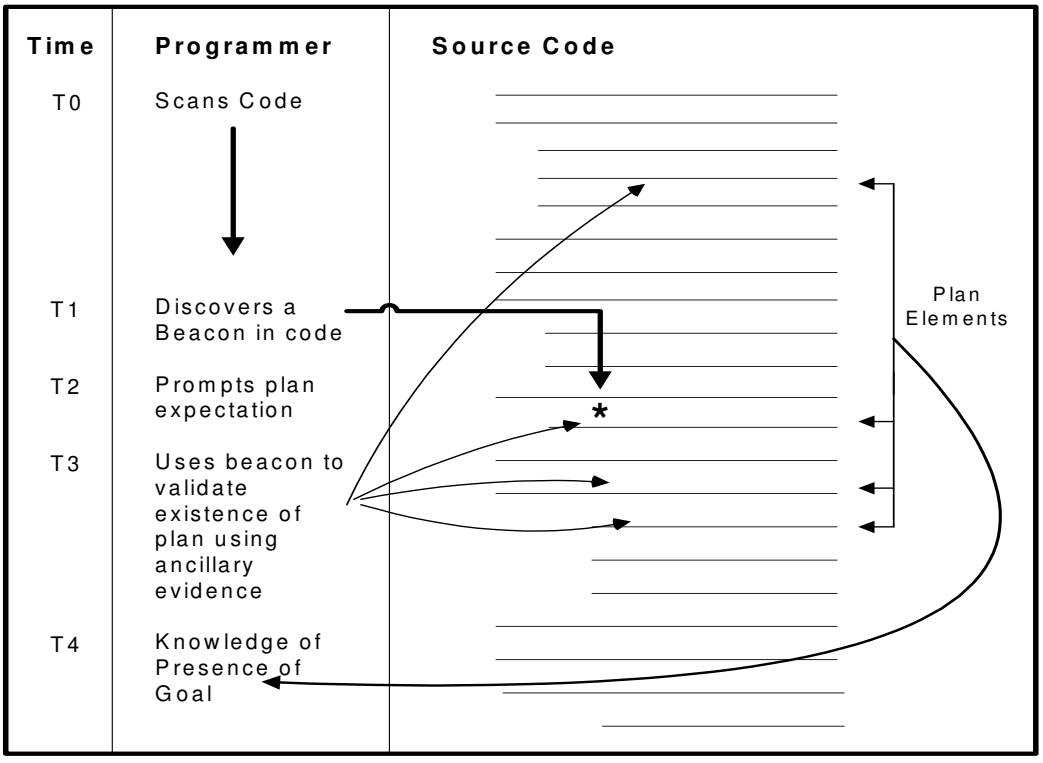

Figure 4 – Opportunistic Recognition of Programming Plans

They propose that programs are composed from programming plans, which have been adapted to meet the needs of the specific problem. They also believe that these plans are composed in accordance with the rules of programming discourse, i.e., the "rules that specify the conventions in programming that programmers should adhere to, to communicate with others" (Soloway & Ehrlich, 1984). Programmers initially scan the code (Figure 4, T0) using 'shallow reasoning', for critical line(s) or beacons. These beacons suggest the presence of plans in their source code. After finding the beacon (Figure 4, T1), programmers seek to validate the presence of the goal (Figure 4, T3) by studying the code for additional elements of the plan (Figure 4, T2).

Plans, defined earlier, are also referred to as 'clichés', and are essentially, as Von Mayrhauser (1995) states, "knowledge elements for developing and validating expectations, interpretations, and inferences". Von Mayrhauser further describes plans in terms of *slot types* and *slot fillers*. Slot types can be thought of as generalised templates, which the maintainer can apply to a number of specific problems to create slot fillers. Slot fillers are specific to a particular problem. Slot types can thus be thought of as an abstraction of a possibly infinite collection of slot fillers. An example of a slot type could be a function such as a sort routine, whereas, a slot filler could be a particular implementation of a sort routine, for example a quick sort. Slot fillers are related to slot types via either a *Kind-of* or an *Is-a* relationship.

Soloway & Ehrlich (1984) state that plans correspond directly to the notion of 'schemas' in text comprehension, where schemas are generic knowledge structures that guide the comprehender's interpretations when understanding text passages. In the context of program comprehension, plans also capture the programmer's attention and guide the actual understanding process itself.

Rules of Programming Discourse then, are related to this concept and can be defined as code conventions that aid the recognition, and thus development, of programming plans; in other words, they are used to build stereotypical implementations of goals that fit the mental schema of others. Soloway & Ehrlich's research presents five initial rules of programming discourse: -

1. Variable names should reflect function.
2. Exclude redundant code (i.e. code that will not be executed)

3. Variables initialized by assignment statements should be updated by assignments

4. Code should not be repeated to do the same job

5. An 'IF' statement should not be used when a statement body is guaranteed to be executed only once, and 'WHILE' statements should only be used for repeated execution.

Soloway & Ehrlich (1984) believe that programs are built from both knowledge of plans, and the rules of programming discourse. They attempted to examine the role of this programming knowledge in the comprehension process by carrying out several experiments. Due to the in-depth experience of expert programmers, Soloway & Ehrlich found that expert programmers performed better on plan-like programs, than on unplan-like programs. This they believed was due to their experience in the programming field, and their tendency to recognize the familiar rules of programming discourse. On unplan-like programs, Soloway noticed that experts' performance deteriorated, as they seemed confused by the rule violations, and indeed their performance levels dropped to near the level of the novice programmers.

2.2.5 Schemas

Letovsky (1986) and Detienne (2002), present the concept of 'schemas', as a way to describe the knowledge of expert programmers. A schema can be defined as "a data structure, which represents generic concepts stored in memory" (Detienne, 2002). Schemas contain variables and are instantiated when specific values are linked to the different variables. Somewhat analogous to 'plans', schemas have been used in the areas of artificial intelligence and psychological studies of text understanding (Eysenck & Keane, 2000). According to this approach, the activity of designing software consists of retrieving schemas from memory, suitable for dealing with certain contexts, and instantiating them to produce a program. During software comprehension, programmers activate these schemas from memory, using 'indexes' in the source code and infer certain information, prompted by the schemas invoked (Detienne, 2002). Essentially, 'indexes' are analogous to beacons, or focal lines (Rist, 1986), where they correspond to the focal part of the code, triggering the activation of

schemas and subsequent hypotheses or expectations of what else will be found in the program.

Letovsky's 'plausible slot filling', is analogous to schema-based processing, where the implementation detail encountered by the programmer causes a suitable abstract goal to be activated in the programmer's memory (Letovsky, 1986). This goal is in the form of a 'frame', which contains a number of titled information slots (variables). Each of these slots indicates some characteristic of the goal and can usually be filled by a restricted set of values. The programmer populates one of the slots in this frame with a plausible value from the implementation detail encountered, and subsequently validates this by further exploring this implementation detail.

Soloway and Ehrlich (1984) hypothesise that programmers initially scan the code using 'shallow reasoning', for critical line(s) or beacons, which they use to prompt plans in their mental model. This, again, seems analogous to Letovsky's plausible slot filling, in that implementation detail causes a suitable abstract plan to be activated in the programmer's memory. Programmers seek to confirm or validate the plan of that goal they have generated from the code by filling the slots.

2.2.6 Domain Goals & Clichéd Implementations

During comprehension, the programmer may be involved in searching the system for plans, which correspond to specific domain goals. This implies that the programmer has knowledge of the typical goals of the domain modelled by the system and the possible manner in which those goals many be implemented (Von Mayrhauser & Vans, 1997).

2.2 Approaches to Software Comprehension

The process or method, which programmers use in undertaking the comprehension task, is known as the 'assimilation process'. Assimilation makes use of the knowledge base and mental model, discussed earlier, in conjunction with the source code (and any other relevant external representation), to build upon, or revise, the mental model of a system, and thus the programmers' understanding of the program. Although some of these processes (Shneiderman & Mayer, 1979), (Brooks,

1983), have been partially described in the description of the knowledge base, they will be further elaborated upon in this section.

2.2.1 Assimilating Program & Situation Models

Pennington (1986) presents a theory of program comprehension based on the framework of text comprehension, put forward by Kintsch & Van Dijk (1978) and Van Dijk & Kintsch (1987). Pennington suggests that the programmer, during the comprehension process, begins at the source code level, understanding small fragments. These small fragments are composed into larger aggregates, whose purpose is constructed from its parts. In other words, this process initiates at the level of source code statements, and builds up to more macro textual components, eventually leading to an understanding of domain-centred goals. Chunking is the central activity in this comprehension process and is used to develop two different mental models: a program model (at source code level) and a situation model (at domain level). Programmers begin by firstly attempting to form the program model, in which the control blocks (while loops, for loops, if constructs) of the program are the basis for mental 'chunks'. Following the partial construction of this program model, the programmer then begins to create a more domain-oriented model of the system, the situation model. The development of this situation model requires knowledge of the application domain to mentally represent the code in terms of real-world objects and to identify the program's functional hierarchy (Detienne, 2002).

Von Mayrhauser and Vans (1995) propose an integrated theory of software comprehension, combining features of the program and situation models (Pennington, 1987), and the 'top-down' model (Soloway & Ehrlich, 1984). This integrated process or 'meta-model', evolved from the experiments carried out by von Mayrhauser & Vans, which concluded that programmers use a combination of assimilation processes when understanding software (see Figure 5).

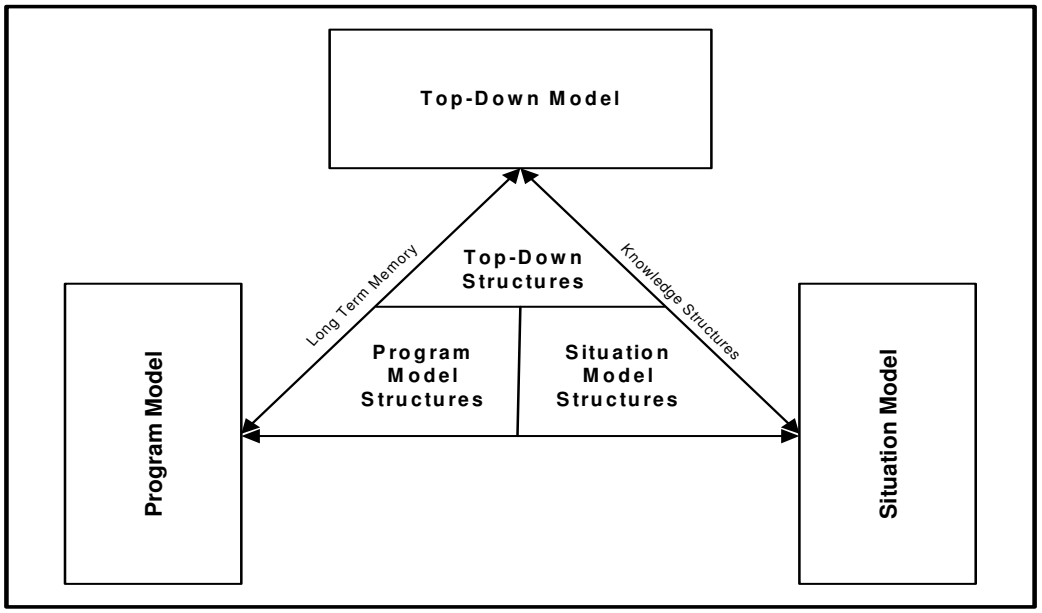

Figure 5 – Von Mayrhauser & Vans (1995) Meta-Model

The integrated model is made up of four main components, namely, the program model assimilation process, the situation model assimilation process, a top-down assimilation process, and a knowledge base. The first three components reflect the comprehension processes of (Pennington, 1987) & (Soloway & Ehrlich, 1984) respectively, while the knowledge base refers to the knowledge required to perform these processes. Programmers may switch between all three approaches at any time during the comprehension process, as indeed, Von Mayrhauser has shown in her empirical work. During a comprehension session, the knowledge base stores existing and newly acquired knowledge for future use.

Von Mayrhauser & Vans state that the top-down model comes into effect predominantly when the programmer is familiar with the program or its application domain. In the presence of unfamiliar code, the programmer switches predominantly to the bottom-up model (although there is evidence to suggest that the true picture can be more complex (O'Brien & Buckley, 2001)). Here, program and situation models are built following a multilevel approach: a preliminary situation model is developed after a partial program model has been formed and both models are then refined with an opportunistic strategy. However, recognition of a beacon during the construction of the program may suggest a high level hypothesis, causing a return to the top-down model.

The integrated meta-model has been used to identify the sequences of activities carried out to accomplish a comprehension goal and to understand how these are aggregated into higher-level processes (von Mayrhauser & Vans, 1996), (von Mayrhauser & Vans, 1994). These can form the basis for identifying information needs during program comprehension and to define useful tool capabilities (von Mayrhauser & Vans, 1993).

2.2.2 Domain Mapping

This process views knowledge as being organised into distinct domains that bridge between the original problem and the final program (Blum, 1989). The assimilation process is one of reconstructing knowledge links between these domains for the programmer (Brooks, 1983). This reconstruction process is theorised to be top-down and hypothesis driven, when the programmer is familiar with the application domain. An initially vague and general hypothesis is refined and elaborated upon based on the programmer's knowledge, information extracted from the program text and other relevant documentation. When the comprehension process is complete, Brooks (1983) claims that a hierarchy of hypotheses will exist with the primary hypothesis at the top, and the subsidiary hypotheses below. Brooks further indicates that hypotheses may fail for one of three reasons. Firstly, the programmer may not be able to find code to bind to a particular subsidiary hypothesis. Secondly, the same code may be bound to two different hypotheses. Finally, there may be some parts of the code that cannot be bound to any hypothesis. Regrettably, Brooks did not carry out any formal empirical evaluation of this theory, although several researchers have since performed empirical studies that support it (von Mayrhauser & Vans, 1995), (von Mayrhauser & Vans, 1997), (Shaft & Vessey, 1995). The main limitation with this theory is that it over-emphasises the 'top-down' approach to comprehension, dismissing other strategies as 'degenerative processes'. It does not take into account, programmers who are inexperienced in the domain, who cannot use 'top-down' comprehension as they are lacking the knowledge to formulate the hypotheses in the first place. Novice programmers therefore seem to resort to a 'bottom-up' process based on the source code, which is not available here. Shaft (1995), in her experiments, makes explicit the high degree of programmer effort spent on bottom-up processing, even when programmers comprehend code from their domains of expertise.

2.2.3 Text Understanding & Problem Solving

Detienne (2002) suggests that there are two contrasting types of approaches to software comprehension: 'text understanding' and 'problem solving'. She proposes that there are three main models to describe the way programmers understand software, namely, the 'functional model', the 'structural model' and the 'mental model'.

2.2.3.1 Text Understanding Approach

The functional model is mostly top-down and schema driven. According to this model, as the programmer reads the program source code (text), they match the content to schemas, using program structure and variables / variable names, as beacons. However, this requires the programmer to know many unique schemas, making this model somewhat irrelevant to programmers who are unfamiliar and inexperienced programming with the domain. Results from Detienne's experiments showed that only experts possess and invoke content rich schemas or 'knowledge schemas' to understand their programs. In their experiments, Soloway and Ehrlich (1984) showed that if the schema's implementation in the code were deliberately made non stereotypical, expert programmers would have much more difficulty in understanding programs than novices. Thus a possible confliction arises here on how expert programmers perceive a non-clichéd program. Some programs fit a lot of schemas, while others, perhaps created by novices, may contain many violations.

The structural method to comprehending software utilises a hierarchical breakdown of the program. Detienne states that according to this structural approach, to understand a program is to construct a network of relations. Two types of model fall into the structural method, namely, the structural schema approach and the propositional network. The structural schema approach shares the 'top-down' orientation with the functional approach. Essentially, structural schemas, or "superstructures", represent the "general structure of stories" (Detienne, 2002), and are activated during the course of reading the program text to guide the understanding process (structural in nature). The propositional network approach, unlike the structural schema approach, is bottom-up in orientation. A 'proposition' can be defined as a unit of information that contains one of more arguments along with a relational term. Propositions are linked among themselves by referential links, and

according to this approach, comprehension depends on identifying the referential coherence among the program's textual elements.

Unlike Pennington's model (Pennington, 1987), these two models (functional & structural) can be directly mapped to functional and object oriented programming as functional programming lends itself to propositional modelling while OO lends itself to situational modelling. This is by far a more plausible method of understanding software.

The third, and final component of this model is the 'mental model' based on (Pennington, 1987), which combines the structural approach with the functional approach. Essentially this is a bottom-up internal representation of the program that considers the problem goals and data flow. The mental model approach to software comprehension means that in order to fully understand a program, one has to construct a detailed representation of the 'situation'. Building this mental model is a time consuming effort, as it is constrained by the limited capacity of working memory. The process, itself, is based on the invocation of domain knowledge, as much as on semantic & general knowledge, such as knowledge schemas.

2.2.3.2 Problem Solving Approach

Koenemann & Robertson (1991) suggest that seeing program comprehension, as a "problem solving activity" is probably a much better overall framework than identifying it as a textual understanding activity. Research by Gilmore & Green (1984), also argues that understanding a program is to solve a problem and Littman et al (1986) suggest that programmers read source code either systematically, or in a non-linear way, reflecting a level of decision making on what parts of the code are relevant to the task at hand.

Littman et al (1986) argue that there are two basic approaches to program comprehension. First there is the systematic approach, where the maintainer examines the entire program tracing through the control flow and data flow abstractions, so as to gain complete understanding of the program. This understanding is made up of static knowledge, where the programmer knows what's in the function, and dynamic knowledge, where the programmer realises the delocalised run-time relationships

involved. These models are completed before any attempt is made to modify the program.

The other approach is the as-needed strategy, where the programmer focuses only on the code related to the particular task at hand and does not study the dynamic relationships in much detail at all. This approach involves the software engineer looking only at code related to a particular problem or task. One limitation of this approach highlighted in Young (1996) is that it can miss some of the dependencies within code fragments. The effectiveness of as-needed comprehension methods then become questionable when using them for defect detection in object-oriented code, which, due to its many features, contains a greater number of dependencies than traditional procedural programs.

However, using the as-needed strategy, subjects can minimise their understanding, only concentrating on the areas of the program that they feel are relevant to the modification task at hand. In contrast to this, subjects using the systematic approach should get a feel for how the modifications may affect the data flow and control flow between program component feeling that they would be required to have this knowledge to 'mesh' the alteration with the entire system. Littman et al's empirical validation, does, however, have one limitation. On the small program used in their experiment, it is clear that the systematic approach is superior, but on large programs this approach is not feasible and an as-needed strategy may need to be employed. Furthermore, the comprehension tasks in this experiment involved programmers debugging code, however, the findings may not apply to contexts outside debugging (Good, 1999).

Gilmore & Green (1984) propose that it is the actual task being carried out, whether it be maintenance or otherwise, which determines the cognitive processes employed, to search for information relevant to carrying out the task. These cognitive processes depend entirely on the relevance of that information to the task at hand. As different programming languages stress different information types, Gilmore & Green compared information notations in procedural & declarative languages, based on a question-answering task. They found that the procedural notation was superior for answering sequential questions, i.e., questions about what happens in a program after some action, X, is performed. On the other hand, the declarative notation was better

for answering circumstantial questions, i.e., questions about what combination of circumstances in a program will cause action, X, to performed. This study shows the importance of the information selection process, and representation, and it is therefore plausible to state that performance is influenced by the notational structure of the language, when the structure is compatible with the demands of the task.

2.2.4 Letovsky's Observations

Letovsky (1986) states that, "the human understander is best viewed as an opportunistic processor capable of exploiting both bottom-up and top-down cues as they become available". In this light, he has developed a theory of software comprehension, based on a more general understanding of the interplay between different human cognition processes. Essentially this implies recognition that a combination of strategies may be employed during comprehension.

The knowledge base of this approach consists of programmer expertise and background knowledge, in the form of application and programming domain knowledge, program goals, plans, and rules of discourse. The mental model essentially encodes the programmer's understanding of the program. Initially, it consists of a specification of the program's goals but later describes the implementation in terms of the data structures and algorithms used. This mental model is organised into three different layers: the specification layer, which describes the goals of the program, the implementation layer, which expresses the lowest level of abstraction, and the annotation layer, which links each goal in the specification layer with its realisation in the implementation layer. The assimilation process describes the evolution of the mental model using information contained in the knowledge base, and if explicit, from the external representation. Assimilation is opportunistic and occurs either top-down or bottom-up depending on the programmer's initial knowledge base.

As Letovsky states, the human understander exploits the varying strategies depending on clues in the available information. Programmers may begin by scanning the code for these beacons (Soloway & Ehrlich, 1984). Once a beacon has been identified, it may trigger a hypothesis, which in turn, may trigger several more subsidiary hypotheses, eventually leading to validation against the code (Brooks,

1983). During this validation, the programmer may opportunistically identify another beacon, which prompts for an alternative hypothesis (Soloway, 1984). The elements of domain knowledge in Soloway's approach are similar to Brooks' domain, and the elements of goal and sub-goals, have a high level of similarly to the task and intermediate level domains that Brooks mentions. Also, Soloway's formal definition of plans, correspond to the relationships between the low-level domain that Brooks mentions.

Letovsky (1986) also carried out an experiment in which maintenance programmers were given a program to modify, and encouraged to think-aloud so that their thoughts could be recorded. From the recordings, he focused on two types of events, namely questions, and conjectures, and developed taxonomies of these events. From the question taxonomy, he identified some of the processing taking place: -

'How Questions' – These questions illustrated that programmers work from expectations down to the implementation expected in the code. For example, "How is the tax receipt calculated"? Here the tax receipt is their expectation and they are trying to find the implementation in the code. Letovsky's 'how questions' and conjectures tie in with Brooks' (1983) hypotheses generation and verification stage, discussed earlier, where hypotheses are matched against the code. These hypotheses can be in the form of a goal-oriented question such as Letovsky's how question, e.g. "how is the tax receipt implemented?"

'What Questions' – These questions illustrated that programmers work from the code to some idea of its purpose. For example, "What does this variable actually do in the code"?

'Why Questions' are questions that again illustrated that programmers work from the code to some idea of its purpose. For example, "Why is this variable in the code"? **'**What' and 'why' questions illustrate that the programmer is working from the source code to domain rationale, in other words, creating a situation model of the system (Pennington, 1987), (Detienne, 2002).

Conjectures were classified in a similar manner to identify their top-down or bottom-up orientation. It was Letovsky's taxonomy of questions that led him to

suggest that a mixture of strategies were employed during comprehension. Part of this cognitive model of program comprehension is basically consistent with the model developed by Brooks. Whereas Brooks emphasises movement from the domain to the implementation approach to reading programs, Letovsky offers convincing evidence that programmers incorporate this process into an opportunistic approach that uses other strategies where applicable.

3. Contextualizing Software Comprehension

The previous sections of this paper examined some of the accepted theories of software comprehension, which attempt to explain how a software engineer goes about the process of understanding code.

Essentially, the process of software comprehension is carried out by people with varying levels of prior knowledge and expertise, and can involve languages with many different characteristics. Just as there is no one definition of 'comprehension' (Sneed, 1995), there is no 'one' all-encompassing model that can summarise the comprehension processes of programmers with varying skill levels and knowledge differences and contexts. Gilmore (1990) suggests that having a wide repertoire of programming strategies available is one of the hallmarks of expert programmers. Therefore, an opportunistic approach to comprehension may be the most plausible, where programmers adopt their comprehension strategies in response to external cues and stimuli.

Empirical studies carried out by O'Brien & Buckley (2001) showed the different comprehension strategies employed by real programmers in an industrial setting. Their studies found three distinct strategies; expectation-based comprehension, where the programmer uses domain knowledge to pre-generates hypotheses, inference-based comprehension, where the programmer infers meaning from clichéd implementation in the code, and bottom-up comprehension, where the programmer works through the code on a line-by-line basis, building up an understanding methodically.

- Using these findings (O'Brien & Buckley, 2001) as a preliminary foundation for future work, research should now begin to probe the 'grey areas' between, and

around, expectation-based, inference-based, and bottom-up software comprehension strategies (see Figure 6), along with the realism of these processes.

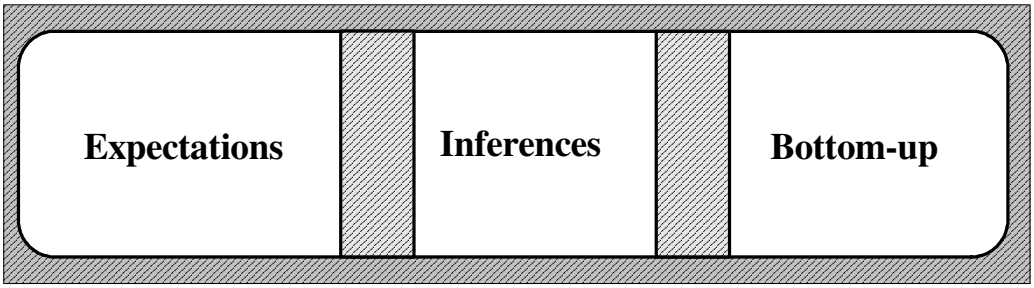

Figure 6 – A Basis For Future Research

Perhaps the reality of these comprehension strategies (grey areas) can be best probed, in a more realistic setting, where the programmer uses the executing system as an external representation, as is very often the case. Very seldom, it is envisaged, do programmers actually sit down with code and understand code for their own sake. Typically the comprehension they undertake is shaped by a wider task context (Littman et al, 1986), (Good, 1999), (Detienne, 2002). During this task, programmers may have an executing software system as a resource, and very little research has been performed on this more realistic scenario. *"Software is inherently dynamic, yet much of the analysis and comprehension processes focus entirely on the static source code of the software"* (Smith & Munro, 2002). It seems intuitive to think that the presence of dynamic representations (executing system) will have strong effects on the comprehension processes employed by software engineers as they attempt to understand a system, and future work should address how representations of the executing system impact the software comprehension process. Some researchers, such as Gugerty & Olson (1986), and Nanja & Cook (1987), have carried out miniature studies, where programmers had access to dynamic representations. However, research in this area remains very limited.

- However, as more subtle comprehension processes are being probed, the analysis used to distinguish between these subtle categories becomes much more involved and intricate. O'Brien & Buckley (2000a) used an augmented version of Shaft's coders manual (Shaft, 1992) as a classification instrument for distinguishing

between expectation-based, inference-based, and bottom-up comprehension. It is envisaged that future research will further enhance this classification schema as these subtle comprehension processes are explored.

A major aspect of future work should be to examine what actually triggers software practitioners' hypotheses in an industrial setting and whether these hypotheses are domain, algorithmic, or structural, in nature. Hypotheses may be triggered from a knowledge of the company's coding conventions, the code mnemonics, the structure of the code, the documentation, or it may be pre-generated – Brooks (1983), for example, suggests that the trigger is pre-generated and that beacons are used to confirm the existence of plans. Finally, the trigger may be execution based (see Table 1).

Trigger based on…						
Cause / **Type**	Coding conventions	Code Mnemonics	Code Structure	Documentation	Pre-generated expectation	System Execution
Domain						
Algorithmic						
Structural						

Table 1 – Identifying the Nature of Hypotheses (Initial Proposal)

This research should also aim to establish where these hypotheses are mapped to in the source code. Essentially, domain hypotheses are primarily pre-generated and usually require an in-depth knowledge of the application domain. O'Brien & Buckley (2001) found that programmers' familiarity with the application domain was associated with a comprehension process that relied upon pre-generated expectations. Using the knowledge of the application domain, programmers were able to pre-generate expectations about typical program goals from that application domain. Algorithmic hypotheses can be defined as hypotheses, which are generated from a [sometimes partial] examination of the code. In other words, having an in-depth knowledge of the programming language semantics, programmers are in a good position to generate algorithmic hypotheses from the actual source code itself. Finally, structural hypotheses are hypotheses generated,

again from an examination of the source code, and in a pilot study carried out by O'Brien & Buckley (2000b) some programmers recognised structural patterns in the code and attempted to draw "structure charts" of the code when understanding it.

Perhaps industrial programmers behave similarly when decomposing the source code into procedures. Future work should build upon research carried out by Von Mayrhauser (1999) and examine the usage of each type of hypothesis made by industrial programmers as they carry out their day-to-day maintenance tasks. One way of doing this is to carry out an empirical study to observe, in situ, routine maintenance tasks by a software practitioner, capturing his/her thought processes and visual environment (screen), using a camcorder. A questionnaire administered to the engineer prior to the study would assess their levels of expertise along with what aspects of the domain under observation, are most familiar to the engineer. Essentially, this 'realistic' study, aims to examine the activities and strategies of industrial programmers to elucidate what they *actually* do in situ, rather than the experimenter presenting them with source code and enforcing time limits, etc. Emphasis should of course be placed on: rich descriptions of the protocols observed in the experiment, the instrumentation used in the experiment (debugging facilities, etc), the reliability of the results, and the validation and replication of the results achieved.

- Another major area of future work is to address the interplay between expectation based, inference-based, and bottom-up, software comprehension strategies, if they prove to be strategies that translate into real domain, by carrying out experiments on individual software practitioners in an industrial setting, as they perform their day-to-day maintenance tasks. This research should also attempt to probe the factors that cause programmers to switch from one comprehension process to another, or to emphasize one over another. There is no agreement among researchers as to what the "correct" model of software comprehension actually is (Good, 1999). Some theories are diametrically opposed: either programmers pre-generate hypotheses and validated them in the code, or they start with the code, understanding it on a line-by-line basis until a full understanding is reached. They cannot do both, unless they subconsciously adopt an opportunistic model as in

Letovsky (1986), (Von Mayrhauser & Vans, 1995). However, there are certain factors, which passively affect the choice of pre-dominant comprehension strategy programmers' employ during cognition, and these are not always intuitive (O'Brien & Buckley, 2001). This should form a major part of future research.

One of the main problems facing novice industrial programmers is their lack of prior programming and domain knowledge. As a result of this, a 'top-down' approach to understanding would prove quite difficult to employ, as the initial stages (i.e. hypotheses formulation) depend on information these programmers do not possess. Instead, they may adopt a bottom-up approach to understanding, where they study the program on a line-by-line basis, building up to an understanding methodically.

If programmers possess an extensive knowledge of the domain at hand, they are likely to be able to generate these hypotheses about program goals and match them against the code, following Brooks' model (Brooks, 1983) of program comprehension. Programmers may also scan through the code in a bottom-up fashion, and after finding a beacon, may generate a new hypothesis about the code, or indeed, detect more information about previously generated hypotheses. Programmers may attempt to piece together all the facts relating to the program, gaining an eventual understanding. In other words, the level of domain knowledge associated with the individual programmer may indeed determine the strategy employed. Other factors that may influence the passive choice of strategy include: having the executing system as an external representation, along with the conventions of the company, such as indentation of the source code and useful comment lines, and finally, the language of the program itself (see Table 1).

Essentially then, future research should have at its heart, an agenda that aims to make empirical studies more justifiably sound, establishing them as an acceptable approach to gaining invaluable insight into the comprehension strategies of industrial programmers. Regrettably, many computer scientists dismiss empirical studies as "ineffective" and some argue that empirical studies are a shallow and artificial view of the nature of programming skill (Sheil, 1981). This pessimism is possibly due to the fact that many studies in the area of software comprehension use students

(novices) as participants in empirical studies. Indeed, this would be justified if their studies examined novice programmers, but in order to bridge the so-called gap between theory and practice, it is imperative that professional, industrial programmers serve as participants, so as to gain insight into the comprehension processes and techniques of 'real-world' programmers. The use of professional industrial programmers' in their naturalistic environment makes results more justifiably sound, and allows for a less constricted experiment design (O'Brien et al, 2001). In an industrial setting, groups of programmers may spend months or longer, on programs containing several thousand lines, which may have originally come from many different sources (Good & Brna, 2003). It is therefore imperative to study industrial software practitioners so as to renew, or at least aim to renew, the trust of practice, and indeed, academia, in empirical studies of programmers.

4. Conclusion

The comprehension of computer programs is a complex 'problem-solving' task. This paper has reviewed the various cognition theories that attempt to model programmers understanding processes. The paper discussed how, for example, programmers may generate hypotheses about the functionality of the code and how 'beacons' in the code build new, or simply refine, hypotheses, during the comprehension session. Most of the theories discussed in this paper, are the result of empirical studies based on observations of programmers during software maintenance and evolution, however, the ecological validity and realism of these studies is questionable. The overall aim of these models is to help to develop better tools and processes that will assist software maintainers with their tasks. However, no 'complete' theory exists that can provide an *all-encompassing model* of programmers comprehension strategies when understanding source code. Early work in this area tended to use undergraduates as participants in empirical work where the task was to understand relatively small pieces of code. Recent work has used industrial software practitioners as participants, however, their comprehension task was chosen prior to the experimental session, by the experimenter. Future work should indeed concentrate on industrial programmers, as subjects, carefully documenting their understanding processes as they carry out their 'day-to-day' maintenance activities. In particular, observational studies should be carried out to assess the role of dynamic

representations (executing system) in the comprehension process as very little research has been carried out to date on this scenario.

Although observational methods of research somewhat sacrifice experimental control, they may provide a richer picture and perhaps, a more realistic one, and accurately reflect how industrial programmers comprehend software, thus bridging the so-called gap between research and practice.

4. Bibliography

Blum, B., (1989), "Volume, Distance, and Productivity", *Journal of Systems and Software,* Vol. 10, pp 217-226

Brooks, R., (1983), "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, Vol. 18, pp 543-554

Burd, L., Munro, M., & Young, P., (2000), "Using Virtual Reality to Achieve Program Comprehension", http://www.year2000.co.uk/munro.html

Davis, S. P., (1993), "Models and Theories of Programming Strategy", *International Journal of Man-Machine Studies*, Vol. 39, pp 237-267

Deimel, L., Naveda, J., (1990), "Reading Computer Programs: Instructor's Guide and Exercises", *Technical Report CMU/SEI-90-EM-3*, Software Engineering Institute, Carnegie Mellon University.

Detienne, F, (2002), "Software Design – Cognitive Aspects", Springer-Verlag London, Ltd., ISBN: 1-85233-253

Eysenck, M., Keane, M., (2000), "*Cognitive Psychology: A Student's Handbook*", 4th Edition, Psychology Press

Gellenbeck, E., Cook, C., (1991), "An Investigation of Procedure and Variable Names as Beacons During Program Comprehension", *Technical Report 91-60-2*, Oregon State University

Gilmore, D. J., Green, T. R. G., (1984), "Comprehension and Recall of Miniature Programs, *International Journal of Man-Machine Studies,* Vol. 21, pp31-48

Gilmore, D. J., (1990), "Expert Programming Knowledge: A Strategic Approach", In Hoc, Green, Samurcay, and Gilmore, (Editors), *Psychology of Programming*, Computers and People Series, Chapter 3.2, Academic Press, London

Good, J., (1999), "Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension", *Ph.D. Thesis*, University of Edinburgh.

Good, J., Brna, P., (2003), "Toward Authentic Measures of Program Comprehension", *Proceedings of the Joint EASE and PPIG Conference*, Keele University

Gugerty, L., Olson, G., 1986, "Comprehension Differences in Debugging and Novice Programmers", Proceedings of Empirical Studies of Programmers, Ablex Publishing, pp 13-27

Kintsch, W., van Dijk, T. A., (1978), "Toward a Model of Text Comprehension and Production", *Psychological Review*, Vol. 85, pp 363-394

Koenemann, J., & Robertson, S., (1991), "Expert Problem Solving Strategies for Program Comprehension", *Proceedings of the SIGCHI Conference on Human Factors in Computing*, ACM Press

Letovsky, S., (1986), "Cognitive Processes in Program Comprehension", *Empirical Studies of Programmers: 1st Workshop*, p 58

Littman D.C., Pinto, J., Letovsky S. and Soloway E.. (1986) "Mental Models and Software Maintenance". *Empirical Studies of Programmers: 1st Workshop*, pp 80-98

Muller, H., (1994), "Understanding Software Systems Using Reverse Engineering Technology", http://www.rigi.csc.uvic.ca

Nanja, M., Cook, R. C., (1987), "An Analysis of the On-line Debugging Process", in Empirical Studies of Programmers, 2nd Workshop, Ablex Publishing

O'Brien, M. P., Buckley, J., (2000a), "The GIB Talk-aloud Classification Schema", *Technical Report 2000-1-IT*, Limerick Institute of Technology, Available on request from authors

O'Brien, M. P., Buckley, J., (2000b), "Pilot Study Refinement of a Software Comprehension Based Experiment", *Proceedings of the 5th Science and Computing Research Colloquium*, Athlone Institute of Technology

O'Brien, M. P., Buckley, J., (2001), "Inference-based and Expectation-based Processing in Program Comprehension", *Proceedings of the 9th International Workshop on Program Comprehension*, Toronto, Canada

Pennington, N., (1987), "Comprehension Strategies in Programming", *Empirical Studies of Programmers: 2nd Workshop*, p 100

Rist, R., (1986), "Plans in Programming: Definition, Demonstration, and Development", *Proceedings of the 1st Workshop on the Empirical Studies of Programmers*, Ablex Publishing, pp 28-47

Robson, D. J., Bennett, K. H., Cornelius, B. J., Munro, M., (1991), "Approaches to Program Comprehension", *Journal of Systems Software*, Vol. 14, pp 79-84

Shaft, T. M., (1992), "The Role of Application Domain Knowledge in Computer Program Comprehension and Enhancement", *Unpublished Ph.D. Thesis*, Pennsylvania State University

Shaft, T. M., Vessey, I. (1995) "The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension", *Information Systems Research*, Vol. 6, No. 3

Sheil, B. A., (1981), "The Psychological Study of Programming", *Computing Surveys*, Vol. 13, No. 1, ACM Press

Shneiderman, B, Mayer, R, (1979), "Syntactic/Semantic Interactions in Programmer Behaviour", *International Journal of Computer and Information Sciences,* Vol. 8, No. 3

Smith, M., & Munro, M., (2002), "Runtime Visualization of Object Oriented Software", 1st International Workshop on Visualizing Software for Understanding & Analysis, Paris, France

Sneed, H., (1995), "Understanding Software Through Numbers: A Metric Based Approach to Program Comprehension", *Software Maintenance: Research and Practice*, Vol. 7, pp 405 - 419

Soloway, E., Ehrlich, K., (1984), "Empirical Studies of Programming Knowledge", *IEEE Transactions on Software Engineering*, IEEE Computer Society, Vol. SE-10, No. 5, September

Van Dijk, T. A., Kintsch, W., (1983), "*Strategies of Discourse Comprehension*", Academic Press, New York

Von Mayrhauser A., Vans A. M., (1993), "From Program Comprehension to Tool Requirements for an Industrial Environment", *Proceedings of the 2nd International Workshop on Program Comprehension, Italy*

Von Mayrhauser A., Vans A. M., (1994), "Dynamic Code Cognition Behaviours for Large Scale Code", *Proceedings of the 3rd Workshop on Program Comprehension*, Washington

Von Mayrhauser A., Vans A. M., (1995), "Program Understanding: Models and Experiments", *Advances in Computers*, Vol. 40, No. 4, pp 25-46

Von Mayrhauser A., Vans A., (1996), "Identification of Dynamic Comprehension Processes During Large Scale Maintenance", *IEEE Transactions on Software Engineering*, Vol. 22, No. 6

Von Mayrhauser A., Vans A., (1997), "Program Understanding Behaviour during Debugging of Large Scale Software", *Proceedings of Empirical Studies of Programmers: 7th Workshop*, pp 157-179

Von Mayrhauser A., Vans A., Howe A.E., (1997), "Program Understanding Behaviour during Enhancement of Large Scale Software", *Software Maintenance: Research and Practice,* Vol. 9, pp 299-327

Von Mayrhauser, A., (1999), "A Coding Schema to Support Systematic Analysis of Software Comprehension", *IEEE Transactions on Software Engineering*, Vol. 25, No. 4

Wiedenbeck, S, (1986), "Beacons in Computer Program Comprehension", *Intl. Journal of Man-Machine Studies*, 25, pp 697-709

Young, P., (1996), "Program Comprehension", *Visualization Research Group*, Centre for Software Maintenance, University of Durham