Key concepts have been developed by researchers in the cognitive sciences to account for the various aspects of knowledge processing. My goal here is to bridge the gaps between the viewpoints of cognitive scientists and software scientists and practitioners regarding knowledge and outline the characteristics of the related concepts in software methodologies and approaches. The benefits are twofold: a better understanding of the cognitive processes involved in software development and an additional scheme for developing new software practices, methods, and tools [1].

Software development is knowledge-intensive. Many concepts have been developed to ease or guide the processing of knowledge in software development, including information hiding, modularity, objects, functions and procedures, patterns, and more. These concepts are supported by various methods, approaches, and tools using symbols, graphics, and languages. Some are formal; others are semiformal or simply made up of key practices. Methods and approaches in software engineering are often based on the results of empirical observations or on individual success stories.

Key concepts have been developed by researchers in the cognitive sciences to account for the various aspects of knowledge processing. My goal here is to bridge the gaps between the viewpoints of cognitive scientists and software scientists and practitioners regarding knowledge and outline the characteristics of the related concepts in software methodologies and approaches. The benefits are twofold: a better understanding of the cognitive processes involved in software development and an additional scheme for developing new software practices, methods, and tools [1].

The mental processing and representation of knowledge are complex activities, and our understanding is still rudimentary and subject to debate [10]. A general concept for describing knowledge is as elusive as ever, though various key concepts have been developed from specific viewpoints in the cognitive sciences. Some of them are derived from the content or structure of knowledge, others from its representation. Here, “knowledge” refers to a permanent structure of information stored in memory. “Knowledge representation” refers to a transitory construction built up in memory for the processing of a specific situation. Table 1 lists the viewpoint corresponding to each key knowledge concept in the cognitive sciences.

There are many ways to define knowledge. One is to consider the way it is stored in human memory. Related studies have identified two types of knowledge—procedural and declarative—and their corresponding memory contents.

Procedural knowledge, including psychomotor ability, is dynamic. Procedural memory stores all the information related to the skills developed to interact with...
our environments (such as walking, talking, typing, and mouse clicking). Knowledge acquisition is based mainly on practice. Procedural knowledge never requires verbal support and is very difficult to describe but, once learned, is rarely forgotten. Such knowledge includes what we call know-how, or knowledge built up through experience. Early designers of expert systems underestimated the complexity of this knowledge concept.

Declarative knowledge, based on facts, is static and concerned with the properties of objects, persons, and events and their relationships. Declarative memory contains all the information that is consciously and directly accessible. Declarative knowledge is easy to describe and to communicate. Declarative memory consists of two types of knowledge—topic, or semantic, and episodic.

<table>
<thead>
<tr>
<th>Key Knowledge Concept</th>
<th>Viewpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedural/Declarative</td>
<td>Knowledge nature of content</td>
</tr>
<tr>
<td>Schema</td>
<td>Knowledge internal structure</td>
</tr>
<tr>
<td>Proposition</td>
<td>Formal knowledge representation</td>
</tr>
<tr>
<td>Chunking</td>
<td>Representing unit of knowledge</td>
</tr>
<tr>
<td>Planning</td>
<td>Managing knowledge structures</td>
</tr>
</tbody>
</table>

Table 1. Key knowledge concepts in the cognitive sciences

Topic knowledge refers to the meaning of words, such as definitions in dictionaries and textbooks. Topic memory is made up of all the cultural structures of an environment and supports the organization of knowledge related to an environment. Such environments exist at various levels, including social, personal, professional, and technical (such as structured analysis and object-oriented).

Episodic knowledge consists of one’s experience with knowledge. Examples include reusing a function, decomposing data-flow diagrams, defining objects from specification requirements, building entity-relation graphs, and documenting programs. Most of these activities are learned through experience once the topic knowledge is obtained from textbooks or courses.

Software development requires topic and episodic knowledge. Difficulties may arise when software developers have only topic knowledge of the application domain, so experience with the knowledge of the application domain may be left out of the software being developed. An example is a well-designed but inappropriate software application. At the coding level, lack of episodic knowledge in the programming language sometimes results in an unduly complex program. Novice programmers have only a limited store of episodic knowledge.

The quality of the software design derived from a methodology can vary according to the designer's episodic knowledge of the methodology. A methodology learned from a book or a crash course is essentially based on topic knowledge. Some methodologies may require more episodic knowledge than others, and the level of episodic knowledge required should be measured or accounted for in some way when evaluating a methodology or the quality of a software design.

The notion of “schema” was first proposed for artificial intelligence by Minsky in 1975 and for psychological studies by Bower et al. in 1979 for describing knowledge structure. The schema concept assumes that knowledge is stored in a human’s memory in a preorganized way. Schemas describe specific links among various knowledge elements. A schema is a generic structure built up from an undefined variety of topics and from episodic knowledge. The topic part of the schema represents objects or events; the episodic part represents temporal or causal links between objects or events [9].

According to the schema concept, our understanding of the world is based on the structure of the knowledge organization. A schema is a knowledge structure made up of variables, or slots, that capture the regularities among objects and events. These slots take into account the properties of objects, typical sequences, or any typical knowledge related to the schema. They often have default values. Schemas have predefined or assumed values for some of their variables. Schemas are rarely fully specified, and values for objects or relations are often assumed. Schemas are also context-dependent [10]. For example, your schema of your operating system represents your memory organization of the related items of topical knowledge, including icons, setup, layout, and menu structure. It also comprises episodic knowledge built up from users’ experience with the operating system, including how to run a program, open a file, use a spreadsheet, listen to CD music, and use email.

The power of the schemas and their default values are illustrated by the following scenario (which includes many schemas):

Daniel opens his computer and edits his homework while listening to music by Mozart. He emails a chapter to his classmate for revision, then plays a game called ABC while waiting for her reply.

You understand this scenario based on your schemas, which are filled with default values based on your personal topic and episodic knowledge related to them. For example, the user's schema of a computer
has a default value corresponding to the operating system for the editing schema; the default value could be any of the many text editors (such as Word, Wordperfect, Frameword, and LaTeX). Your schema of Mozart’s music could be just some classical music or specific Mozart masterpieces. However, you are likely to lack a default value for the schema of the electronic game ABC.

Schema default values can be unexpected yet major components in software development activities, because they are based on the developer’s personal experience with a particular area of knowledge. Schema default values need to be validated. Walkthroughs, reviews, and inspection meetings help validate or define the default values of the various schemas used. Although schema validation is carried out implicitly most of the time, it might be rewarding to base these meetings on explicit schema default-value validation. In such cases, each schema and its corresponding validated variables are identified.

Schemas have been used to study text understanding and software comprehension [2]. Some software methodologies explicitly promote the use or creation of schema; an example is software patterns.

AI also uses the notion of schema as a data structure representing a concept stored in computer memory. The theory behind the schema in AI uses the notions of frames, scripts, and plans. Natural schemas (not those used in AI) are fuzzy concepts. Schemas are all embedded and usually a mix of subschemas, which are also mixtures of subschemas. At some point, we reach basic, or primitive, schemas, leading to the notion of the "proposition."

Knowledge formulation is based on atomic components described in terms of propositions and predicates. A proposition is the smallest unit of knowledge constituting an affirmation as well as the smallest unit that can be true or false. Propositions are discrete representations of knowledge elements and seem to be interconnected in memory based on their shared arguments. According to some authors in AI, mental models are defined by propositional representations [3].

The theoretical hypothesis concerning the cognitive structure of human information systems states that, at a certain level, information is organized in propositional form. Many psychologists studying the representation of meaning in memory believe that the propositional representation is the dominant representation in the human brain and that propositions have three principal functions:

- They can represent any well-specified information from which it follows that propositions form a general mechanism for representing knowledge.
- They preserve the meaning but not the form of a statement or sentence.
- They naturally support reasoning and inferences.

Based on a model of text comprehension, it has been estimated that the working human memory can handle from one to four propositions. A proposition is a formal representation of knowledge. Software development based on formal specifications relies on the propositional representation of knowledge, which is applicable mainly to well-defined problems.

A problem is well-defined if the initial state, the goal, and a set of possible operations are available to reach the goal from the initial state. An academic problem, for example, is a typical well-defined problem, formulated from defined knowledge and requiring students to select the right set of operations for its solution. It might ask you to: Write a program in C to implement a first-in-first-out data structure. Solutions to such problems are characterized by a search in the memory for existing algorithms that may provide the answer. The ability to solve well-defined problems is acquired through study. These problems are intellectual exercises and formalized easily.

An ill-defined problem does not have a well-specified goal because many goals may be acceptable. In the same way, the cognitive approaches used to solve ill-defined problems cannot be defined clearly because there might be many ways to solve them [12].

Software design problems usually belong to the family of ill-structured problems. Their solutions are acceptable in varying degrees and are rarely either absolutely correct or absolutely incorrect. The design task structures a problem by finding the missing information or creating new information and using it to specify new goals within the problem-knowledge space [4]. Software design is generally a mixture of ill- and well-defined problems. The specification and the
design of the algorithms or the system architecture often constitute an ill-defined problem type; translation of the detailed design into programming code is more of a well-defined problem type.

The nature of the problem is not defined in an absolute way, depending somehow instead on the solver’s level of experience. A novice may find a problem ill-defined, while an experienced designer considers it well-defined, because a well-defined goal for reaching the solution is available. A problem is perceived as being more or less complex, depending on the existing goal definition in the mind of the solver. The perception of complexity raises the problem of software complexities in the planning activity.

We humans intuitively understand that good design emerges naturally from the formal specification of a well-defined problem. However, less obvious is that formal specifications are appropriate for ill-defined problems. Formal specifications are based on propositions. Recall that propositions have three functions (listed earlier); of these, the first and the third do not apply to ill-defined problems. The information is clearly well-specified, and reasoning and inference are not the dominant mental activities required to solve ill-defined problems. The answer to the everlasting debate on the appropriateness of formal specifications for software development may lie in the nature of the problems to be solved. Formal specification seems to be more appropriate for well-defined problems.

Another component of the mental process is the amount of knowledge available for immediate processing. Psychologists use the concept of chunks to account for the limited amount of knowledge that can be handled by the human mind at any given time. Chunks are general and do not refer to the information content of the knowledge. It is well known (since Miller’s classic experiment in 1957) that short-term memory, or working memory, has a limited capacity and can typically process only 7±2 chunks at a time.

For example, it is more difficult to memorize the letter combination BMI LMU than IBM UML. The second sequence is, however, no more difficult to memorize than computer/Unified Modeling Language. In the first sequence, each letter is a chunk (six chunks, for a total of six letters); in the second, each group of letters is a chunk (two chunks, for a total of six letters); and in the third sequence, each group of words is a chunk (two chunks for a total of 31 letters). A chunk is a unit of information whose significance varies with the individual reader. For example, UML may not be easier to remember than LMU for someone unfamiliar with the UML object-oriented methodology; in such a case, IBM UML is composed of four chunks (1+3).

Software methodologies based on encapsulation, information hiding, modularization, abstraction, and even the divide-to-conquer approach all deal with the chunking phenomenon. Successful methodologies based on icons, graphic symbols, and reserved words are naturally limited to the chunk number for the simultaneous use of elements in working memory. Extensive computer-aided software engineering (CASE) tools or methodologies requiring over-chunking by users (too many features to remember simultaneously) are likely to meet with a lack of approval. Since chunks do not refer to information content, they are a measure of the unrelated knowledge that can be processed naturally.

Making Plans
Software development involves processing a large amount of information distributed over many knowledge domains that are more or less contiguous with fuzzy frontiers that are intertwined most of the time. The limited capacity of the human mind’s working memory cannot keep track of all the information from all the knowledge domains visited. Plans are therefore needed to manage the knowledge. Plans are knowledge representations used to organize knowledge based on various criteria and to guide the tasks to be done by the mind. The properties of plans are anticipation and simplification. Anticipation accounts for the expected results associated with a plan and are based on experience. Plans can be a set of subgoals defining the main steps to be reached before a final goal is achieved. Plans are not necessarily procedural, and each subgoal does not necessarily correspond to a well-defined activity. Plans have three main characteristics [6]:

- A heuristic nature. Plans efficiently guide mental activity toward the most promising avenue based
on the knowledge available without a detailed analysis of the situation.

- Optimal use of memory. Plans keep only critical properties of objects or events by making abstractions of all nonsalient details associated with the activity being carried out.
- Higher control level. Plans enable the emergence of an activity that cannot be derived from the detail of the activity being processed.

The following scenario illustrates how plans guide the design process. The mental structure of the designer is in a “local” knowledge state; local means that the amount of knowledge available at any given time for immediate processing by the brain is limited to the capacity of the working memory. The designer must therefore move continuously from the local state to another state of knowledge. This move can be done in a completely arbitrary fashion or can be based on plans. Arbitrary or heuristic approaches can be associated with a dreaming activity or limited self control in the state of mind entered into. Software designers would rather (hopefully) rely on planned activities that can be either rigorous and systematic or opportunistic.

A systematic planning approach is when designers believe they have access to all the knowledge required to do the task. It has been observed in studies related to the psychology of programming that, for example, experts adopt a planning mechanism based on a breadth-first approach, while novices, who often rely on their understanding of programming languages, adopt a depth-first approach. Designers actually follow well-structured plans as long as they find nothing better to do. When knowledge is not readily available, some explanatory mechanisms are required. The explanatory process is called “opportunistic,” because at various points in the process the designer makes a decision or takes action depending on the opportunities presented. The decisions are motivated by earlier decisions and are not the product of a well-planned process.

Designers progress from a systematic planning activity to an opportunistic one with the evolution of the design, a process that is not always balanced. Design rationales are especially useful when opportunistic planning has occurred, because they capture the information on which decisions are based.

Serendipitous planning occurs when designers try to group together or integrate a set of decisions or plans into a single coherent plan. Grouping together means that partial solutions are recognized at various levels of detail and are combined. Software reuse is suitable for serendipitous planning.

Developing plans depends on designers’ experience with the design solution and their ability to associate existing plans. It has been suggested by researchers in the psychology of programming that lack of experience increases a design’s variability and then contributes to the software’s complexity. Expert knowledge is organized in a more abstract and deeper way. Resulting plans are based on situations already seen, rather than on trial-and-error exploration. Studies on planning have shown that expert plan structures have four abstract characteristics:

- Hierarchical with multiple levels;
- Explicit relationships between levels;
- Based on basic schema recognition; and
- Well connected internally.

Planning is one of the human brain’s most powerful natural activities, although the mental mechanisms involved are not fully understood and cannot be fully automated. Early methodologies and software tools have sought to define and enforce some planning activities based mainly on hierarchical top-down development, an ingenious approach that generated great expectations but little success.

Some CASE tools are artificial guides for planning activities. The following partial list of desired methodologies or CASE tool features is based on our current understanding of mental processes and is divided into two sections—the first helpful in planning activities, the second helpful in representational activities:

- Helps organize mental activity;
- Enables deviation from or even abandonment of plans; never imposes fixed hierarchical planned activities;
- Supports a return to an original plan, never assuming or imposing it;
- Enables work at various levels of detail and abstraction;
- Helps manage the limits of human memory by making various levels of knowledge available simultaneously;
- Maintains traces of abandoned or interrupted tasks or plans for easy, spontaneous return.
- Generates visual representations adapted to the designer’s level of experience and to the various viewpoints expressed;
- Presents the solution’s constraints;
- Enables easy change in the representation level;
- Helps build representations;
- Helps create the design;
- Captures the design rationale; and
- Outlines plan structures and strategies.
Conclusions
Software development is the processing of knowledge in a very focused way. We can say it is the progressive crystallization of knowledge into a language that can be read and executed by a computer. The knowledge-crystallization process is directional, moving from the knowledge application domain to software architectural and algorithmic design knowledge, and ending in programming language statements.

Software engineers have developed methods, practices, and tools to ease the knowledge-crystallization process. And cognitive scientists have studied the properties of knowledge from various points of view. Our purpose is to merge these two approaches.

New directions in software engineering may result by considering established views of knowledge structures and representations from the cognitive sciences. Each knowledge concept presented here illustrates a feature of the mental processing of knowledge. These concepts are derived from observations and are applicable to any mental activity, including software development.

Cognitive scientists derive their understanding of knowledge through their observation of experts and novices at work and from their controlled experiments. Software engineers have made little use of these approaches, however, and few methodologies or CASE tools are derived from documented observations or controlled experiments.

An immediate benefit for software engineering is to account for the known characteristics of mental knowledge processing. Some methodologies or CASE tools could be improved, or at least be made to not work in a counterproductive way by interfering with the brain's natural processing of knowledge. It should then be easy to identify the knowledge viewpoints targeted by a component of a method or a function of a tool.

Experience plays a major role in any knowledge-related activity. Psychologists recognize a distinct structure, called “episodic,” in human memory that accounts for experience. Any project leader knows the value of experience. Psychologists also know that knowledge processing by an expert is quite different from a novice's knowledge processing. Software engineers rarely define the level of experience required to use a methodology or a tool, and some software advertisements claim no experience is needed. Is such a tool useful to an expert?

It is important to distinguish between the knowledge structures supporting understanding (schemas) and the mechanisms used to organize that knowledge (plans). Software engineers have placed a great deal of emphasis on documenting the final representation of the knowledge structure, or the source code. But the documentation of the plan has only recently been introduced through the design rationale, which documents the process through which the knowledge is structured or crystallized.

Software complexity, software quality, and software metrics may find common ground if the level of opportunistic planning in a given task can be measured. Such a measure would be a sign of the stability, or the quality, of the design and reflect the designer's experience in a particular knowledge domain.

Software development can be improved by recognizing the related knowledge structure or representations, including building schemas, validating schema default values, acquiring topic knowledge, requiring appropriate episodic knowledge, performing planning activities, applying formal specifications (encoding knowledge into a propositional form) to define problems, and having the appropriate tools to manage the chunking phenomenon.

References

Pierre N. Robillard (robillard@rgl.polymtl.ca) is a professor in the Department of Electrical and Computer Engineering at the École Polytechnique de Montréal in Montréal, Canada.

This work was supported in part by grant A0141 from the Natural Sciences and Engineering Research Council of Canada. Additional support was provided by the Applied Software Engineering Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 1998 ACM 0002-0782/99/0100 $5.00