A Theory of Program Comprehension

Joining Vision Science and Program Comprehension

Yann-Gaël Guéhéneuc

PTIDEJ Team

École Polytechnique de Montréal

and Université de Montréal

Montreal, Quebec, Canada

guehene@iro.umontreal.ca

October 13, 2008

In theory, there is no difference between theory and practice. But, in practice, there is.

Jan L. A. van de Snepscheut

Abstract

There exists an extensive literature on vision science, on the one hand, and on program comprehension, on the other hand. However, these two domains of research have been so far rather disjoint. Indeed, several cognitive theories have been proposed to explain program comprehension. These theories explain the processes taking place in the software engineers' minds when they understand programs. They explain how software engineers *process* available information to perform their tasks but not how software engineers *acquire* this information. Vision science provides explanations on the processes used by people to acquire visual information from their environment. Joining vision science and program comprehension provides a more comprehensive theoretical framework to explain facts on program comprehension, to predict new facts, and to frame experiments. We join theories in vision science and in program comprehension; the resulting theory is consistent with facts on program comprehension and helps in predicting new facts, in devising experiments, and in putting certain program comprehension concepts in perspective.

1 Introduction

Joining vision science and program comprehension provides a theoretical framework to explain how software engineers understand programs and, thus, to explain known facts, to predict new facts, and to set up experiments on program comprehension.

In the recent year, the domain of cognitive informatics as emerged to study the internal processing mechanisms of the human brain and their applications in computing. Cognitive informatics is intrinsically multi-disciplinary and unites researchers in several domain of research such as cognitive science, cybernetics, and software engineering. In particular in software engineering, it could bring interesting advances that could explain the mechanisms trough which software engineers understand, write, and debug programs. Therefore, it encompasses the domain of program comprehension.

Program comprehension is the domain of software engineering that seeks to explain how software engineers understand programs [34], *i.e.*, how they obtain a mental representation of a program structure and function [26]. Facts and laws of program comprehension lie at the heart of almost every software related activities, from development to maintenance, deployment, and use. Many studies on program comprehension have been published in the literature. However, understanding program comprehension requires more than just knowing facts; it requires a theory.

Our research concerns situations in which expert software engineers readily recognise well-known patterns (*any* kind of patterns) in a program model, while novice software engineers only perceive their constituents. Figure 1 shows a subset of a UML class diagram. (We cloud class names for the sake of explanation). The question is: What is the structure and function that an expert software engineer recognises instantly in that diagram and that a novice software engineer does not? (The curious reader may take few seconds to study the class diagram.) An expert software engineer could recognise that the classes follow the solution of the Composite design pattern [13, p. 163], thus assigning the function of representing part–whole hierarchies, while a novice software engineer only sees a hierarchy of four classes, some operations, and an aggregation relationship. (Compare with the solution of the Composite design pattern in Figure 4). We attempt to explain this difference in comprehension between expert and novice software engineers with a theory.

Existing cognitive theories[1], such as Brooks' [6], von Mayrhauser's [42], Pennington's [30], Soloway *et al.*'s [38], provide explanations on the short-, long-, and working-memories used, on the cognitive internal processes at play, and on the internal and external knowledge incorporated and constructed during program comprehension. Other authors, such as Minsky [24], Rich and Waters [31], and Soloway [37] theorise the cognitive internal representation of knowledge through the concepts of frames, plans, and chunks.

[1]We use the term "theory" instead of the occasional term "model" (see for example von Mayrhauser's integrated *model* [42]) to prevent confusion with program models.

However, no existing theory of program comprehension explains well the difference between expert and novice software engineers, in particular wrt. the use of patterns.

Our approach to explain the previous difference recognises the importance of sight during program comprehension and consists in describing the information flow from graphical program models (texts or diagrams) to cognitive internal processes and memories, using vision science. Vision science is an interdisciplinary domain of cognitive science, which provides a framework for understanding vision in terms of phenomena of visual perception, the nature of optical information, and the physiology of the visual nervous system [28, p. 5]. It does not "analyse the sociocultural basis of the staggering amount of functional information people learn about familiar" items[2] but focuses "on the more perceptually relevant question of how sighted people manage to perceive an [item]'s functional significance by looking" [28, p. 409]. Theories of vision science explain the capabilities of the human visual system to acquire visual information and to provide this information to cognitive internal processes and memories. Therefore, we develop a theory of program comprehension including the processes of acquiring and comprehending information through sight, by drawing inspiration from theories of vision science and of program comprehension. This vision–comprehension theory relates vision processes, cognitive internal processes, and memories, during program

[2]We use the term "item" instead of the more common term "object" to distinguish between *visual* objects and objects in object-oriented programming languages.

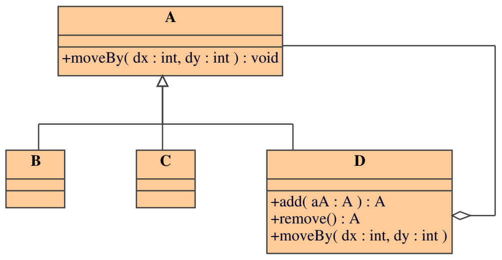
A
+moveBy(dx : int, dy : int) : void
B
C
D
+add(aA : A) : A
+remove() : A
+moveBy(dx : int, dy : int)

Figure 1: What is the structure and function that an expert software engineer recognises instantly in this subset of a UML class diagram of a well-known program and that a novice software engineer does not? How does the recognition happen?

comprehension, and brings together data and methodology of program comprehension and vision science. We show that this theory is consistent with known facts, we explain certain trends in program comprehension, and we propose possible theoretical and experimental falsification studies.

In Section 2, we introduce related work on theories of software engineering, of program comprehension, and of vision science. In Section 3, we present the rationale for developing a theory of program comprehension. In Section 4, we detail the theory and its philosophy, premises, context, and scope. In Section 5, we show that the theory explains known facts, while in Section 6 we discuss trends in program comprehension. In Section 6.4, we propose falsification experiments. Finally, in Section 7, we conclude and introduce future work.

2 Related Work

2.1 What are Theories?

A theory is an integrated set of statements—hypotheses—about the principles that organises and explains known facts and that makes predictions about new facts possible. It must be an internally consistent minimal set of hypotheses from which to derive explanations of known facts and testable predictions of new facts. A theory helps in understanding a domain by explaining a list of "Why is it so?" questions [11, p. 7] [27].

2.2 Theories in Software Engineering

The domain of software engineering possesses theories but existing theories [11] do not explain well known facts, due to the relatively recent existence of this domain, the complexity of the phenomena, and the lack of agreed upon formalisms and notations. Endres and Rombach [11] gather many facts, laws, and theories. However, we disagree with the authors calling their findings "theories". Indeed, their theories are more like laws, because they only explain one (or few) facts at a time, they do not generalise well, and they use many hypotheses with respect to the principle of Ocam's razor.

Empirical studies are recognised as essential to understand phenomena with which software engineering deals, thanks to the work of precursors, such as Vic Basili [5]. Empirical studies are based on the classical cycle: observations (facts), laws, theories. Many facts have been recorded through empirical studies and some laws have been proposed. Yet, few theories have been derived to explain these facts and laws and to predict news facts. Some researchers argue that current empirical studies focus too much on generalisation and not enough on theory building [17].

Another reason for the inadequacy of theories is the lack of frameworks in which to set up experiments and with which to interpret results. Indeed, theories are an invaluable help in setting up experiments to observe facts (dis)proving the laws and theories. (For example, since Albert Einstein proposed his theories of quantum physics, physicists used this framework to set up experiments to prove or to disprove the theories, to obtain and to explain new facts.)

2.3 Theories in Program Comprehension

We decompose the domain of program comprehension in three lines of research: (1) acquisition of data from programs, for example through static or dynamic or documentation analyses; (2) study of the broader context in which program comprehension takes place, *i.e.*, the software engineers' activities, the organisational and social contexts of their activities; and, (3) development of theories based on the understanding of the software engineers' use of extracted data in a given context.

2.3.1 Data Acquisition

The rich literature on program comprehension focused early on the problems of obtaining data from software artifacts (static and dynamic data, features, documentation and other repositories), see for example [1,39]. It also tackles the means to represent and to communicate this data, using various techniques from text-based editors to 3D interactive dynamic environments, such as [36]. This literature is essential to understand what kind of data software engineers have at their disposal to comprehend programs.

2.3.2 Context

Less work study the contexts in which the program comprehension activity takes place. Murphy *et al.* [25] attempt to distinguish, to describe, and to identify recurring patterns in the software engineers daily activities. Although not related to program comprehension explicitly, their work brings insight in the program comprehension activity, because program comprehension is part of all but the most basic software engineering activities. Thus, this line of research is important to generalise claims in program comprehension. Indeed, generalising claims is difficult because no complete taxonomy of the software engineers' activities exist. However, the authors do not frame their experiments in a particular theory of software development and, thus, run the risk to perform unfocused experiments.

2.3.3 Theories

Few theories of program comprehension have been proposed in the literature. We introduce two theories which are compared in more details with our theory in Section 6. The first theory, proposed by Brooks [6], describes program comprehension as a process of building a sequence of knowledge domains, bridging the gap between problem domain and program execution. A succession of knowledge domains describes a software engineer's comprehension of a program.

The second theory, developed by von Mayrhauser [42], is an integrated theory describing the processes taking place in the software engineers' minds during program comprehension, as a combination of top-down and bottom-up comprehension processes, working with a common knowledge base. This integrated theory accounts for the dynamics of forming and of abstracting a mental representation of a program.

These precursor theories are invaluable and we draw much inspiration from their insights. However, none of these theories explain and use the processes of acquisition of the information by the software engineers through their senses.

2.4 Theories in Vision Science

Vision science is the domain of computing science interested in the understanding of people's vision system. Vision science collects facts on vision, formulate laws from these facts, and devise theories explaining these laws and facts. With these theories, vision scientists have been able to predict new facts successfully and to refine their theories.

Vision science possesses many theories to explain colour vision, spatial vision, perception of motion and events, as well as eye movements, visual memory, and visual awareness. To the best of our knowledge, Palmer's book presents the most complete and in-depth coverage of vision theories, cast in the information processing paradigm [28]. For the sake of brevity, we introduce theories of vision science along with relevant references while presenting the vision–comprehension theory in Section 4.

3 Rationale of the Theory

Our theory recognises software engineers as human beings. We believe that the physical and cognitive characteristics of software engineers are important because they use all of their senses and cognitive capabilities when they comprehend a program, in particular sight. Indeed, when comprehending a program, software engineers read documentation and source code and they visualise all kinds of program models. The preeminence of sight is corroborated by the extensive literature on visualisation for program comprehension. Yet, to the best of our knowledge, no previous theory of program comprehension recognised the importance of sight explicitly.

We build on vision science and on program comprehension theories to propose a theory of program comprehension accounting for the acquisition and for the use of visual information by software engineers. This theory is based on theories in vision science but does not contradict existing theories of program comprehension, such as Brooks' [6] and von Mayrhauser's [42]: It extends these existing theories to provide explanations on the acquisition of information.

Section 2 highlights the importance of theories to explain known facts and to predict new facts. In addition, a theory is useful to frame efforts to automate the program comprehension activity. Indeed, as Marr points out, once a "theory for a process has been formulated, algorithms for implementing it may be designed, and their performance compared with that of the human" processor [23, p. 331]. Thus, our theory could help in devising (semi-)automated algorithms to comprehend programs and to help software engineers in comprehending programs.

Finally, our theory will help in devising experiments to answer questions related to the program comprehension activity through sight. Without theories on program comprehension, "the danger is that questions are not asked in relation to a clear [theory]." [23, p. 349] and, thus, provide answers lacking focus and generality.

4 Vision–Comprehension Theory

4.1 Philosophy

The program comprehension activity requires information contained in a program source code and documentation. But this information is not sufficient for comprehending a program. The tasks at hand (why is the program being studied), the context of the activity (where, for what purpose is the program studied), and the experience of the software engineers (with software engineering, program comprehension, and the studied program) are required additional sources of information. Thus, software engineers contribute information during the program comprehension activity. We take a *constructivist* stance, like Floyd [12] for software development, in which comprehension is constructed from external and internal information, without emphasising formalisation at the expense of communication, learning, and evolution.

4.2 Premises

We cast our theory within the information processing paradigm, in which the human brain is seen as a computational processor. This paradigm is built on the similarity between cognitive psychology and computers and belongs to the objectivist tradition [19, p. 99].

We use the meta-theoretical analysis of the information processing paradigm proposed by Palmer and Kimchi [29], which makes different assumptions on the informational description, the recursive decomposition, and the physical embodiment of cognitive processes in the human brain: Cognitive processes, such as visual perception or program comprehension, are processes transforming input information in output information. They can be decomposed in a number of cognitive processes linked by a flow diagram. They are embodied in the behaviour of the *physical* human brain in which they take place.

Thus, assume that software engineers focus their attention on the program model that they comprehend and we describe visual perception and the program comprehension activity with processes acting on different representations of information from the program model.

We also consider a theory of reinforcement learning [40] to explain the behaviour of certain processes with respect to the amount of processing required to perform the program comprehension activity.

4.3 Context

We develop our theory in the context of a growing need to understand program comprehension to reduce maintenance cost and to develop tools to support program comprehension, see for example the extensive literature on software visualisation.

Our theory relates to on-going research on alternative visualisation techniques to represent program models graphically, for example using adjacency matrices [14] or 3D representations [36], to map and to ensure the traceability of mappings between source code and higher-level abstractions, such as design patterns [2] or features [3].

4.4 Scope

We limit the scope of our theory to software engineers engaged in a program comprehension activity. Software engineers must be using sight in normal functional conditions to comprehend program models, other modalities are not accounting for (although they might influence the activity). The program models can be represented in any form, either textual, graphical (bi- or tri-dimensional), including static, dynamic, and–or other kinds of data.

4.5 Definition

We decompose the recognition of items during program comprehension in several processes acting on different representations[3] of a program model. Figure 2 shows the sequence of processes and the flow of data among these.

Although we believe that the following theory provides a source of explanations for the program comprehension activity, beyond its details, this theory is interesting for bridging vision science and program comprehension and for integrating early stages of comprehension (vision) with later stages (cognition [6, 42], see also Section 6).

- **Retinal Image.** The program comprehension activity begins with a retinal image of a program model which a software engineer is observing. The retinal image may originate from any program model, either a textual representation, a UML-like diagram…
- **Image-based Stage.** The retinal image is analysed to extract spatial features from its structure, such as edges, lines, and textures. This process produces a set of visual elements: edges, bars, and blobs [23].
- **Surface-based Stage.** The visual elements from the previous process are interpreted in terms of visible surfaces in a 3D space. The process produces local pieces of oriented surface [22].

[3]We use "representations" to denote cognitive internal representations of a program model and "models" to denote program models, such as code source text, UML diagrams.

[3]Boxes represent image representations, circles are processes, arrows describe information flow, dotted lines show decompositions. Absence of representations between processes does not preclude existence but simplifies the flow.

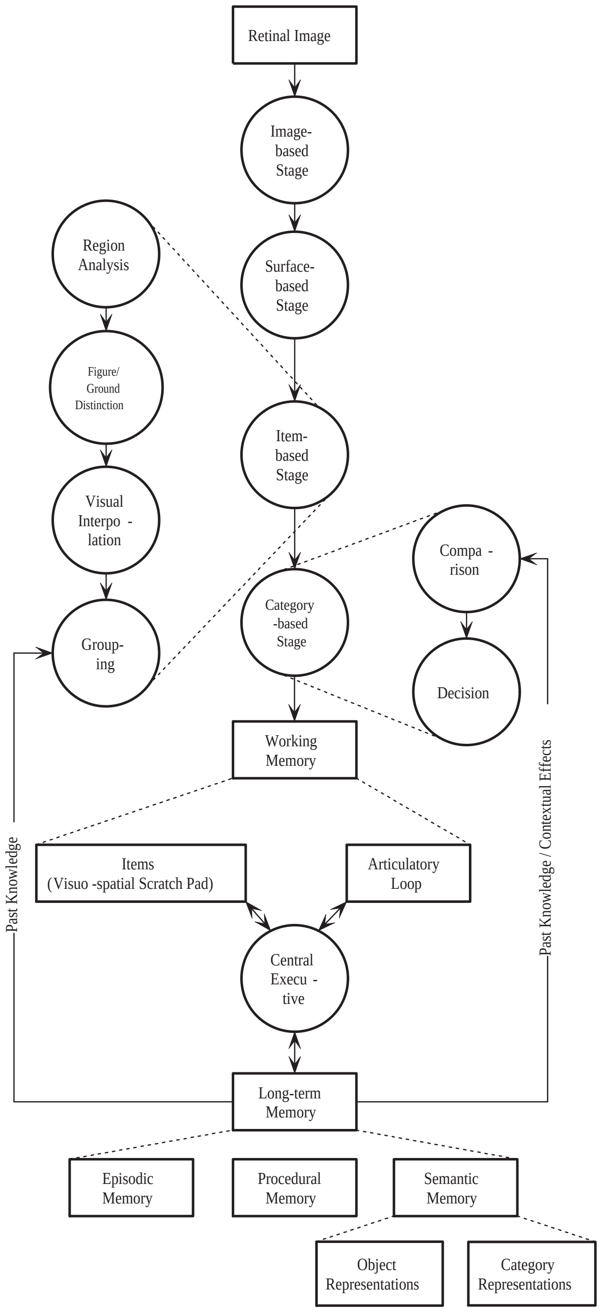

Figure 2: Theory[3] of program comprehension through vision.

- **Item-based Stage.** This process concerns the perceptual organisation of the visual elements (including local pieces of oriented surface) in items, see [28, p. 254]. We decompose this process in four sub-processes:

  - **Region Analysis.** This sub-process analyses visual elements to identify regions in the image structure. A region is a bounded 2D area that constitutes a subset of the image. Different principles direct region analysis, such as connectedness, segmentation, and texture segregation.
  - **Figure/Ground Distinction.** Once regions are identified, this sub-process decomposes regions in two categories: figures and ground. Figures are parts of the image close to the observer, with a bounded contour defining their shapes. Ground is the part of the image farther from the observer, extending behind and not shaped by a contour.
  - **Visual Interpolation.** This sub-process interpolates the hidden parts of identified regions from their visible parts to allow the visual system to perceive partly-occluded visual elements.
  - **Grouping.** This sub-process groups previously identified visual elements in coherent items. It uses laws of grouping, such as proximity, similarities of colour, size, orientation, and common fate. It also uses past knowledge stored in long-term memory to

perform the grouping: From now on, the reader shall always see the use of the Composite design pattern in Figure 1.

- **Category-based Stage.** Grouping visual elements in items is not yet sufficient to allow comprehension. The items must be categorised to distinguish their different functions. We follow the theoretical approach of indirect perception of function, which decomposes in two sub-processes: Comparison and Decision. Following Rosch's theory [33], we assume that memory stores exemplars of items, represented by prototypes, with which to assign identified items to categories.
  - **Comparison.** This sub-process uses past knowledge and contextual effects to compare identified items with known items, accounting for the context in which items are identified. Items may belong to several categories (or to no category).
  - **Decision.** Then, this sub-process decides the categories to which the items belong, typically using a maximum-over-threshold rule.
- **Working Memory.** The previous processes produce a representation in which the various items composing the program model have been identified and categorised. This representation is handled by the Working Memory, which is similar to short-term memory but possesses an important internal structure [4]. The Working Memory decomposes in a Central Executive process interacting with the Visuo-spatial Scratch Pad, the Articulatory Loop, and the Long-term Memory.

– **Central Executive.** The central executive process performs cognitive tasks, such as comprehension, problem solving, and memorisation tasks. It works closely with the two following memories.

– **Visuo-spatial Scratch Pad and Articulatory Memories.** This visuo-spatial scratch pad stores visual/spatial information while the articulatory loop stores verbal information. Other kinds of memories may exist for other modalities. These memories play the role of cache with respect to long-term memory, being faster and easier to access by the Central Executive.

– **Long-term Memory.** The central executive process also interacts with long-term memory. Long-term memory is composed of three memories: episodic, procedural, and semantic, which we describe in the context of program comprehension.

– **Episodic Memory.** The episodic memory stores information on items and events part of the software engineer's life history. It stores domain and functional knowledge learned by the software engineer during apprenticeship and program knowledge learned during program comprehension.

– **Procedural Memory.** The procedural memory stores information about skills and procedures to perform actions. It contains the software engineer's knowledge about programming and strategies for program comprehension [42].

- **Semantic Memory.** Finally, the semantic memory stores information concerning knowledge of concepts, such as in an encyclopedia. This memory stores the visual appearances of prototypical objects and categories, such as architectural, design, and implementation prototypes. It also stores knowledge domains [6] (acquired through sight or built by the Central Executive).

This theory describes the program comprehension activity from a Retinal Image generated by a graphical representation of a program model to the Central Executive, which performs the comprehension activity. Before discussing the theory in Section 6, we use it to explain some known facts.

5 Explanations of Known Facts

5.1 Why are Patterns so Important?

Patterns, such as idioms [7], design patterns [13], or architectural patterns [20], have been quickly and widely adopted by the software engineering community. In particular, the design patterns described by Gamma *et al.* [13] stirred much interest in the software engineering community as a way to document and to reuse good practices in design [7].

The development of a "pattern community" led to a large body of literature on pattern categorisation, formalisation, application, and identification. In this literature, many claims about patterns have been made, in particular regarding the usefulness of patterns to ease program comprehension. The claim on the usefulness of patterns is the main assumption (although often implicit) in research on pattern recovery, in particular design pattern recovery, including work by this author. This claim states that patterns ease program comprehension because software engineers recognise immediately certain patterns in program models as performing functions. Although this claim is sensible, it has never been explained theoretically. We provide an explanation, with our theory, on the usefulness of two kinds of patterns as examples: idioms and design patterns.

```
final Iterator iterator = aList.iterator();
while(iterator.hasNext()) {
    final Object o = iterator.next();
    ...
}
```

Figure 3: Example of the Iterator idiom.

5.1.1 Idioms

Idioms are patterns at the programming language level. Recovery of idioms, *i.e.*, grouping program statements in idioms, has been subject of many work very early, for example the Programmer's Apprentice [31].

The code excerpt in Figure 3 shows the Iterator idiom of Java. A software engineer, who knows this idiom, would recognise the function of the statements instantly and would focus on the use of the list elements rather than on the means to iterate over these elements. We consider three cases to explain the recognition of the function of the statements:

- **The Software Engineer Does Not Know the Idiom.** A software engineer who has never been confronted to the Iterator idiom would be able to extract from the textual representation shown on Figure 3 its different statements through the Image- and Surface-based Stages. Without previous knowledge of the idiom, Long-term Memory would be unable to provide past knowledge to the Grouping and Comparison sub-processes. The statements composing the representation would be analysed by the Central Executive, which would infer and store their function in Semantic Memory, as Object Representation.

- **The Software Engineer Does Not Know the Idiom but Has Encountered its Use.** The software engineer would extract the statements through the Image- and Surface-based Stages. During the Item-based Stage, the Grouping sub-process would group the statements related to the idiom, based on information on the idiom previously stored as an Object Representation. The Category-based Stage would use this information to compare the grouping with previous groupings, to decide their similarity, and to retrieve their function. Information on the function of the statements (along with information on the statements themselves) would go to Working Memory, thus easing the comprehension activity by supplying directly the Central Executive with the function of the statement. The Central Executive, on seeing the same group another time, may promote the group from Object Representation to Category Representation [40].

- **The Software Engineer Knows the Idiom.** Again, the software engineer would extract the statements through the Image- and Surface-based Stages. The Grouping sub-process would group the statements related to the idiom, based on the information on the idiom stored in the Semantic Memory as Category Representation. The Category-based Stage would use the grouping and information from the Category Representation to assign a function immediately, thus easing the use of the grouping during program comprehension by the Central Executive.

Thus, we explain the usefulness of idioms: The knowledge of idioms eases program comprehension by providing the Central Executive with the function of a group of statements directly, without requiring further processing from the Central Executive to identify its function.

5.1.2 Design Patterns

Design patterns, like idioms, are claimed to ease program comprehension. They have been specifically targeted towards software engineers to provide "good" solutions to recurring design problems [13]. To the best of our knowledge, there exists no attempts to prove or disprove these claims but through case studies, see for example [43].

We see design motifs—solutions of design patterns—as prototypes that can be used in the progrram comprehension activity of micro-architectures—subsets of a program design—directly. We believe that design motifs are identical or closed to the prototypes in a software engineer's memory. A software engineer who knows design motifs would recognise their structure and function directly, such as in Figure 1 with the Composite design pattern. We consider three cases to explain the comprehension of micro-architectures:

- **The Software Engineer Does Not Know Design Patterns.** A software engineer with no knowledge of design patterns would extract the constituents of the micro-architecture presented in Figure 1 through the Image- and Surface-based Stages. Long-term Memory could not provide the Item- and Category-based Stages with past knowledge and, thus, Working Memory would perform a time- and resource-consuming analysis of the constituents to identify their function.

- **The Software Engineer Knows an Unused Design Pattern.** A software engineer with knowledge of design patterns not used in the micro-architecture is in a position similar to the one above. The information provided by the Long-term Memory to the Grouping and Comparison sub-processes of the Item- and Category-based Stages would help neither in grouping the constituents nor in categorising the micro-architecture. Again, the Central Executive would be responsible for inferring the function of the constituents.

- **The Software Engineer Knows the Used Design Pattern.** Knowledge of the design pattern which motif is used to implement a micro-architecture decreases the work load on the Working Memory. A software engineer would extract the constituents of the micro-architecture through the Image- and Surface-based Stages. The Grouping sub-process would group these constituents using the knowledge of the similar prototypical design motif. Then, the Comparison sub-process would use this information and the knowledge on the prototype to provide the function of the micro-architecture to the Working Memory directly, thus easing program comprehension.

Thus, we can explain the quick and wide adoption of design patterns and the attention paid to their literary form: The synthetic, prototype-based, descriptions of design patterns are usable directly in the Item- and Category-based Stages of program comprehension through sight, decreasing the work load on the Working Memory and, thus, easing the program comprehension activity.

5.1.3 Practical Considerations

The statement composing an idiom or the constituents of a micro-architecture similar to a design motif might be scattered across a graphical program model and can be even in non-displayed parts of the model. It is important to identify the constituents forming these patterns and to bring these constituents visually together to help software engineers in using design motifs to comprehend the model.

Kosslyn [18] performed several experiments validating the picture metaphor: These experiments revealed a highly linear relationship between response times and distances between pairs of items in an image. Thus, graphically grouping constituents of patterns help in understanding their functions through the Grouping and Comparison sub-processes as well as navigating in the program model by minimising the dwell time of eye movements among constituents. Kosslyn's experiments and our theory are consistent in justifying experimentally and theoretically the benefits of software visualisation techniques, in particular UML-like diagram layout algorithms (for example [10]) and alternative visualisation techniques (see Subsection 5.4)

5.1.4 Conclusion

Our theory explains the importance of patterns: Patterns reduce the amount of processing required by the Central Executive to comprehend the functions of subsets of program models. It provides explanations on the quick and wide adoption of patterns by practitioners due to their usefulness during the program comprehension activity.

Also, our theory provides explanations on the advantages of expert software engineers over novice software engineers: Expert software engineers have read and understood much larger quantities of information related to program implementation (source code) and design (models such as UML class diagrams). They increased the number of prototypical patterns stored in their Semantic Memory. Our theory provides a theoretical basis to the idea that reading and understanding other software engineers' program is beneficial.

Finally, our theory provides explanations on the importance of "good" graphical program models. Graphical models are important because they facilitate the Grouping and Comparison sub-processes. It enforces the idea of standardised models of source code (either textual or graphical), because these models are closer to the prototypical models (in kinds and in forms) stored in Semantic Memory.

5.2 Why are Packages and Composite States so Important?

Packages in graphical program models, such as UML class diagrams, and composite states in UML state diagrams are believed to ease program comprehension. Serrano *et al.* [35] propose a study on packages for modelling data warehouses and Cruz-Lemus *et al.* [8] a study on program comprehension using composite states, which tend to show that, indeed, packages and composite states ease program comprehension.

We can explain theoretically the results of these studies in a similar fashion as we explained the importance of patterns. Packages (composite states, respectively), when used adequately, are groups describing sets of functions. A software engineer, while studying a program model, would extract information on the packages through the Image- and Item-based Stages. The Item- and Category-based Stages would not need to use past knowledge on individual constituents because packages are recognised as a single item (in opposition to the package constituents, that would be recognised as a set of items to be grouped). Information on the packages would go to the Working Memory. The individual constituents of the packages would never be processed independently, thus reducing the required processing resource and time and easing program comprehension.

However, the usefulness of packages and of composite states has not yet been successfully proven empirically. We believe that absence of experimental proofs results from a lack of theoretical framework in which to cast experiments. Our theory provides an experimental framework in which to devise experiments based solely on studying composite states and packages during program comprehension, wrt. other factors such as tool support, software engineers' experience.

5.3 Why do Bounded Information ease Program Comprehension?

Gail *et al.* [25] present a study of software engineers' tasks, based on the monitoring and recording of the tasks during daily work. Tasks are composed in task structures. The authors claim that task structures help in reducing software engineers' time and effort to find relevant information by reducing the amount of displayed data, by bounding queries, and by performing queries automatically. However, they do not cast their claims within a theoretical framework of program comprehension to justify what information software developers need, when they need it, and how they use it. Thus, they do not explain *why* well-defined task structures would actually ease software engineers' activities. The benefits of task structures remains hypothetical until generalisable empirical proofs are provided.

We can explain why the use of task structures could indeed reduce software engineers' time and effort to perform their program comprehension activity by presenting data in a readily processable way—close to prototypes that software engineers have in memory. Moreover, the data embedded in task structures could be used along with the idea of parallel processing and visual pop-out [41], to provide software engineers with visual clues on important information.

5.4 Why do Alternative Visualisation Techniques ease Program Comprehension?

Ghoniem *et al.* [14] realise an experiment on adjacency matrices, assessing their usefulness for different comprehension activities with respect to node–link diagrams.

An adjacency matrix represents the relationships between sets of items. An adjacency matrix is only limited by the characteristics of the display, while a node–link diagram is limited also by the characteristics of the displayed network.

The authors conclude that adjacency matrices have several benefits (in time and in accuracy wrt. graph size and density) over nodes–links diagrams to estimate numbers of nodes and links, to find connected, specified, and neighbouring nodes and links, but not to find a path between nodes.

We explain the differences between adjacency matrices and nodes–links diagrams in three ways. First, estimation and finding activities requiring software engineers to distinguish nodes are performed with adjacency matrices easily because nodes–links diagrams require more processing during the

Item- and Category-based Stages, in particular during Region Analysis, due to the cluttering of the diagrams.

Second, finding specific nodes and links is faster and more accurate with adjacency matrices because software engineers only need to recognise the node number, which is performed easily by the visual system or to put in correspondence two nodes, which is performed by the visual system efficiently.

Finally, finding a path between nodes is difficult for the visual system using adjacency matrices because the software engineers must focus their attention on several different parts of the matrices, thus losing their focus in the transition from one node to another, while in nodes–links diagrams, they follow links among nodes, thus keeping their focus.

6 Discussions

6.1 Theoretical Bias

We postulated that we ease program comprehension by reducing the amount and the time of processing required by software engineers' minds to comprehend program models. Thus, we assume *from the start* that the information processing paradigm is correct because we assume that the program comprehension activity depends on the characteristics of the *processes* used to perform this activity.

This assumption biases our theory because we develop our theory *using* the information processing paradigm. However, to the best of our knowledge, the information processing paradigm has never been successfully challenged yet. In the domain of vision science, Gibson [15] is the principal opponent of the information processing paradigm. We could envision program comprehension theories based on Gibson's approach to vision but we favour building on the extensive literature related to the information processing paradigm.

6.2 Level of Abstraction

The information processing paradigm used to describe the processes involved in program comprehension also helps in defining the concept of level of abstraction precisely.

Given the information obtained through a program comprehension activity using the vision system, the level of abstraction of a program model qualifies the amount of processing required to obtain this information from the model. Thus, for a same information provided to the Central Executive, a program model at a low-level of abstraction requires more processing (including from the Central Executive) than a program model at a higher-level of abstraction.

We illustrate this definition of the level of abstraction with Figures 1 and 3. A typical implementation of the Composite design motif uses the Iterator idiom in D.moveBy(int,int) to apply recursively the A.moveBy(int,int) method on the stored instances of A (or of its subclasses). A software engineer cannot obtain easily information on the structure and function of the Composite design motif through the statements in Figure 3 (even with its complete implementation), while this information is obtained directly from the program model in Figure 1. The program model in Figure 1 is at a higher level of abstraction than that in Figure 3 because the former requires less processing to obtain a given information.

Previous work providing program models at a higher level of abstraction than source code through static, dynamic, or semantic analyses could be assessed using this definition of the level of abstraction for comparison and for evaluation of their algorithms and of the resulting program models.

6.3 Comparison with Existing Theories

Our theory does not infirm Brook's precursor theory of program comprehension as a succession of refined knowledge domains. Indeed, our theory explains the acquisition of knowledge domains by software engineers through vision and their uses by the Grouping and Comparison sub-processes and by the Central Executive, from the Semantic Memory. Thus, we build on Brooks' theory by incorporating seamlessly the idea of knowledge domain as part of the Semantic Memory. A detailed study of their differences and similarities could lead to their fusion in one complete theory.

Von Mayrhauser's theory [42] ("integrated model" based on previous theories, such as Litovsky's) is a complete theory of program comprehension, accounting for top–down and bottom–up comprehension strategies, and program knowledge. Our theory does not challenge von Mayrhauser's theory because it provides explanations on the acquisition of program models through vision *before* any comprehension strategies is applied. It uses ideas from von Mayrhauser's theory to account for the behaviour Central Executive and for the use of past knowledge in the Grouping and Comparison sub-processes. Also, we do not assume that the objective of comprehension is to understand a program completely, as highlighted by von Mayrhauser. Thus, the two theories are complementary and explain different parts of the program comprehension activity. They could be merged to detail the complete program comprehension activity.

6.4 Falsifications

Attempts to falsify a theory are important for its refinement.

6.4.1 Theoretical Falsification

We propose three possible directions for falsifying our theory theoretically. First, our constructivist stance could be disputed and other philosophical stances could be used for software development and program comprehension, which would frame these activities better.

Second, we cast our theory in the information processing paradigm, but other paradigms could be envisioned to build program comprehension theories. Although we do not believe that an ecological paradigm, such as proposed by Gibson [15], could be used in program comprehension easily, research is needed to study other paradigms to progress in the understanding of program comprehension.

Finally, the flow of data among processes, in the definitions of the (intermediary) representations and of the processes, must be detailed. We adapted theories of vision science in one consistent whole without overloading the resulting theory with detailed explanations. We favoured a more descriptive approach to parallel the program comprehension activity and the theory. Further studies could prove parts of (or all of) our theory wrong, thus advancing the understanding of the program comprehension activity.

6.4.2 Experimental Falsification

Experimental falsification of our theory includes devising experiments to disprove, for example, the flow of data among processes, the use of past knowledge during the Grouping and Comparison sub-processes, or the usefulness of these sub-processes for program comprehension. Experimental falsification could be based on introspection: observation of one's own conscious experience while comprehending programs. However, cognitive psychology rejects introspection [9] as a valid method of investigation (except through the "think-aloud" protocol [21]). Thus, we propose to *observe* software engineers engaged in program comprehension.

Progress in non-intrusive monitoring of human behaviour allows to monitor external activities of software engineers involved in a program comprehension activity. We plan to use video-based eye tracking systems to assess the pertinence of design pattern identification for program comprehension. Video-based eye tracking systems allow to follow software engineers' eye movements while looking at a program model [16]. We will assess design pattern identification, on the one hand, and the use of these patterns during program comprehension, on the other hand, to improve current techniques of design pattern identification and of representations of micro-architectures similar to design motifs.

First, we will use a video-based eye tracking systems to study software engineers' eye movements on the constituents of program models (for example, UML-like class diagrams) and the dwell time on individual constituent (classes or relationships) to assess the identification of patterns. The hypothesis is that novice software engineers do not identify patterns in program models and, thus for a set of identical comprehension tasks, navigate through the models quite differently than expert software engineers. The

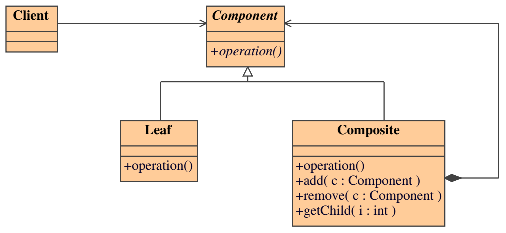

Figure 4: Original class diagram of the Composite design pattern [13, p. 163].

expected conclusion is that expert software engineers identify patterns in program models, while novice software engineers do not, thus needing more time to comprehend models.

Second, we will study the use of identified patterns. Given a program model and a set of clearly identified patterns, we will compare software engineers' eye movements when performing different program comprehension tasks, distinguishing software engineers with and without pattern knowledge. The hypothesis is that pattern knowledge eases the program comprehension activity by allowing expert software engineers to focus their attention (eye movements) on constituents *inside* or *outside* of the identified patterns instead of navigating among all their constituents. The expected conclusion is that expert software engineers benefit from their knowledge of patterns by focusing their attention relatively to these patterns. This conclusion could also be in agreement with the idea of intentionally-ignored information validated by Rock and Gutman [32] experimentally.

7 Conclusion and Future Work

We proposed a theory of program comprehension to explain the differences between expert and novice software engineers, in time and in effort, to perform program comprehension, in particular with patterns. This theory draws on the extensive literature in vision science and on previous theories of program comprehension. It is cast in the information processing paradigm and explains known facts on program comprehension, such as the importance of idioms, patterns, packages, composite states, and alternative graphical models, in terms of internal cognitive representations and processes. It provides a framework for concepts of program comprehension, such as level of abstraction, and for falsification studies, both theoretically and experimentally.

This theory is a step to explain the program comprehension activity and may be inadequate for some aspects of the activity. We humbly hope that our theory will foster new research as stepping stone towards more accurate explanations. Future work includes a detailed study of the internal cognitive representations used by the processes, their forms and their characteristics. It also includes a closer comparative study (and possibly fusion) of our theory with von Mayrhauser's and Brooks'. We shall also perform falsification experiments using video-based eye tracking systems. Finally, other modalities should be studied, such as hear. Other future work also includes developing our theory to integrate negative facts found in the program comprehension domain, *i.e.*, facts that contradicted the hypotheses of the researchers that studied them. Integrating such “negative” facts would help develop a theory that is more robust to contradicting facts.

Acknowledgements

The author thanks Naouel Moha, Victor Ostromoukhov, and Duc-Loc Huynh deeply for the fruitful discussions.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1^{st} edition, January 1974.

[2] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *Proceedings of the* 16^{th} *Conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.

[3] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: A novel approach and a case study. In Tibor Gyimóthy and Vaclav Rajlich, editors, *Proceedings of the* 21^{st} *International Conference on Software Maintenance*, pages 357–366. IEEE Computer Society Press, September 2005. Best paper.

[4] Alan D. Baddeley and Graham Hitch. Working memory. In Gordon Bower, editor, *The Psychology of Learning and Motivation*, volume 8, pages 47–90. Academic Press, 1974.

[5] Barry Boehm, Hans Dieter Rombach, and Marvin V. Zelkowitz. *Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili.* Springer-Verlag, 1^{st} edition, September 2005.

[6] Ruven Brooks. Using a behavioral theory of program comprehension in software engineering. In Maurice V. Wilkes, Lazlo Belady, Y. H. Su, Harry Hayman, and Philip Enslow, editors, *Proceedings of the* 3rd *International Conference on Software Engineering*, pages 196–201. IEEE Computer Society Press, May 1978.

[7] James O. Coplien. Idioms and patterns as architectural literature. *IEEE Software Special Issue on Objects, Patterns, and Architectures*, 14(1):36–42, January 1997.

[8] José A. Cruz-Lemus, Marcela Genero, M. Esperanza Manso, and Mario Piattini. Evaluating the effect of composite states on the understandability of UML statechart diagrams. In Lionel Briand, editor, *Proceedings of the* 8th *International Conference on Model Driven Engineering Languages and Systems.* Springer-Verlag, October 2005.

[9] Knight Dunlap. The case against introspection. *Psychological Review*, 19:404–413, 1912.

[10] Holger Eichelberger and Jürgen Wolff von Gudenberg. On the visualization of Java programs. In Stephan Diehl, editor, *Proceedings of the* 1st *international seminar on Software Visualization*, pages 295–306. Springer-Verlag, May 2002.

[11] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering.* Addison-Wesley, 1st edition, March 2003.

[12] Christiane Floyd. *Human Questions in Computer Science*, chapter 1, pages 15–27. Springer Verlag, March 1992.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[14] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In Matt Ward and Tamara Munzner, editors, *Proceedings of the 10^{th} symposium on Information Visualisation*, pages 17–24. IEEE Computer Society Press, October 2004.

[15] James Jerome Gibson. *The Perception of the Visual World*. Greenwood Publishing Group, hardcover edition, December 1950.

[16] Yann-Gaël Guéhéneuc, Stefan Monnier, and Giuliano Antoniol. Evaluating the use of design patterns during program comprehension – experimental setting. In Giuliano Antoniol and Yann-Gaël Guéhéneuc, editors, *Proceedings of the 1^{st} ICSM workshop in Design Pattern Theory and Practice*. IEEE Computer Society Press, September 2005. In the pre-proceedings.

[17] Magne Jørgensen and Dag I. Sjøberg. Generalization and theory-building in software engineering research. In Stephen Linkman, editor, *Proceedings of the 8^{th} international conference on Empirical Assessment in Software Engineering*, pages 29–36. IEEE Computer Society Press, May 2004.

[18] Stephen M. Kosslyn. Scanning visual images: Some structural implications. *Perception and Psychophysics*, 14:90–94, 1973.

[19] Roy Lachman, Janet L. Lachman, and Earl C. Butterfield. *Cognitive Psychology and Information Processing: An Introduction.* Lawrence Erlbaum Associates, Publishers, 1^{st} edition, June 1979.

[20] Nicole Lévy and Francis Losavio. Analyzing and comparing architectural styles. In Raul Monge and Marcello Visconti, editors, *Proceedings of the* 19^{th} *international Conference of the Chilean Computer Science Society.* IEEE Computer Society Press, November 1998.

[21] Clayton Lewis. Using the "thinking-aloud" method in cognitive interface design. Technical Report RC9265, IBM T.J. Watson Research Center, 1982.

[22] David Marr. Representing visual information. In Allen R. Hanson and Edward M. Riseman, editors, *Computer Vision Systems*, pages 61–80. Academic Press, 1978.

[23] David Marr. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information.* Henry Holt & Company, 1^{st} edition, June 1982.

[24] Marvin Minsky. A framework for representing knowledge. Technical Report Memo 306, MIT AI Laboratory, June 1974.

[25] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Čubraniś. The emergent structure of development tasks. In Andrew P. Black, editor, *Pro-*

ceedings of the 19^{th} *European Conference on Object-Oriented Programming*, pages 33–48. Springer-Verlag, July 2005.

[26] Raquel Navarro-Prieto. *The Role of Imagery in Program Comprehension: Visual Programming Languages.* PhD thesis, University of Granada, 1998.

[27] Allen Newell. You can't play 20 questions with nature and win. In W.G. Chase, editor, *Visual Information Processing.* Academic Press, 1973.

[28] Stephen E. Palmer. *Vision Science: Photons to Phenomenology.* The MIT Press, 1^{st} edition, May 1999.

[29] Stephen E. Palmer and Ruth Kimchi. The information processing approach to cognition. pages 37–77. Lawrence Erlbaum Associates Publishers, 1986.

[30] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Journal of Cognitive Science*, 19(3):295–401, July 1987.

[31] Charles Rich and Richard C. Waters. *The Programmer's Apprentice.* ACM Press Frontier Series and Addison-Wesley, 1^{st} edition, January 1990.

[32] Irvin Rock and Daniel Gutman. The effect of inattention on form perception. *Journal of Experimental Psychology: Human Perception and Performance*, 7:275–285, 1981.

[33] Eleanor H. Rosch. On the internal structure of perceptual and semantic categories. In Timothy E. Moore, editor, *Cognitive Development and the Acquisition of Language*, pages 111–144. Academic Press, 1973.

[34] Spencer Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.

[35] Manuel Serrano, Rafael Romero, Juan Carlos Trujillo, and Mario Piattini. The advisability of using packages in data warehouse design. In Fernando Brito e Abreu, Coral Calero, Michele Lanza, Geert Poels, and Houari A. Sahraoui, editors, *Proceedings of the* 9^{th} *workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 118–128. CRIM, Montreal, July 2005.

[36] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In Pedro Sousa and Jürgen Ebert, editors, *Proceedings of the* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 30–38. IEEE Computer Society Press, March 2001.

[37] Elliot Soloway. Learning to program = Learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, September 1986.

[38] Elliot Soloway, Jeannine Pinto, Stanley Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communication of the ACM*, 31(11):1259–1267, November 1988.

[39] Diomidis Spinellis. *Code Reading: The Open Source Perspective.* Addison Wesley, 1^{st} edition, May 2003.

[40] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1^{st} edition, March 1998.

[41] Anne Triesman. Preattentive processing in vision. *Computer Vision, Graphics, and Image Processing*, 31(2):156–177, August 1985.

[42] Anneliese von Mayrhauser. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.

[43] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *Proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.