Program Understanding Models: An Historical Overview and a Classification

Eric Harth and Philippe Dugerdil
Geneva School of Business Administration, HES-SO, Tambourine 17, 1227 Carouge, Switzerland

Keywords: Program Comprehension, Comprehension Models, Understanding Strategies.

Abstract: During the last three decades several hundred papers have been published on the broad topic of "program comprehension". The goal was always the same: to develop models and tools to help developers with program understanding during program maintenance. However few authors targeted the more fundamental question: "what is program understanding" or, other words, proposed a model of program understanding. Then we reviewed the proposed program understanding models. We found the papers to be classifiable in three period of time in accordance with the following three subtopics: the process, the tools and the goals. Interestingly, studying the fundamental goal came after the tools. We conclude by highlighting that it is required to go back to the fundamental question to have any chance to develop effective tools to help with program understanding which is the most costly part of program maintenance.

1 INTRODUCTION

As more and more source-code is inherited from the past, often still used in production environment, and since a high turn-over is common in the software development industry, program maintenance is often done at a very high cost (Tilley, 1997). Trying to understand programs written by others involves many different skills, the knowledge of several domains, as well as dedicated strategies and integrated processes in order to analyse and discover the original programmer's intent. During the past few decades, many studies have been undertaken to identify what mental models were used when writing the programs and how these models can be recovered by reading the source code by studying the program documentation or by analysing the program's behaviour.

In this paper we intend to summarize the most common models identified by the research community during the last three decades as well as the strategies and techniques used while understanding legacy programs. The referenced research works will be classified into different time periods since we realized that their authors targeted different aspects of program understanding over time. The contributions of this paper is first to propose a synthesis of more than 30 years of research in this field. Second, it is the identification of three periods of time among which the research works can be classified. Last, it is to summarize the perspectives under which "program understanding" has historically been studied and to identify the remaining questions to be solved. Finally, the referenced models we present in this work are formatted using a common graphical modelling syntax allowing to compare the models with each other's.

2 DOCUMENT STRUCTURE

In section 3 we present the five most common understanding strategies identified during the last three decades. Section 4 provides an explanation of the three main periods we observed with respect to the topic addressed by the researchers in program understanding. The following sections 5, 6 and 7 present some of the emblematic work in each of the periods. Section 8 contains a discussion of the perspectives under which "program understanding" has been studied over the years and the remaining questions to be answered. Section 9 concludes the paper.

3 UNDERSTANDING STRATEGIES

As proposed by Exton (Exton, 2002) comprehension strategies may roughly be categorized into five

Harth, E. and Dugerdil, P.
Program Understanding Models: An Historical Overview and a Classification.
DOI: 10.5220/0006465504020413
In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT 2017)*, pages 402-413
ISBN: 978-989-758-262-2

generic approaches addressing different needs and involving different representation techniques. All strategies are derived from the "Constructivism Learning Theory" which states that learner is the central part of the learning process, not the studied material (Hein, 1991). This theory also suggests that the learning process, considered a heuristic procedure acquired through individual experiences, cannot be standardized since each individual learns differently at a different pace. Yet, the author noted that some recurring understanding strategies were commonly involved in all these disparate learning processes, especially within the software domain.

3.1 Bottom-up Strategy

This is the simplest approach used when a developer is not familiar with the analysed software. This strategy starts by reading the source code statement by statement, grouping information together (chunking) to form more abstract information. Grouping again and recursively these chunks into more abstract chunks gives, at the end, an abstract overview of what the software does in very generic terms. This strategy is qualified as "bottom-up" since the abstraction process start from low-level items (the source-code) up to a global description of what the software does and why it is written so.

3.2 Top-down Strategy

When a developer has already some prior knowledge of a specific domain, he can make high-level assumption about the program, even without reading a single line of code. Starting from the GUI, the program's name or the documentation, he can generate hypotheses about the program's purpose and validate or refine them by reading some code fragments. With a recursive descent he will make hypotheses on more specific features, down to concrete source code items. This strategy is qualified as "top-down" since the strategy starts from very generic assumptions down to more specific ones, until code fragments are matched.

3.3 Hybrid Strategy

This approach – originally named *knowledge-based strategy* by Exton (Exton, 2002) – borrows ideas from bottom-up and top-down strategies and mixes them according to contextual and opportunistic needs. If the developers are already familiar with some of the business concepts manipulated in the code, they will select a top-down strategies. But sometimes they will encounter unfamiliar code structures requiring bottom-up strategies. This opportunistic strategy is called hybrid since it merges the two previous strategies into a single one.

3.4 As-Needed vs Systematic Strategy

When the analysed software is large, developers usually do not try to understand the whole system. Rather, they will limit the time spent on a single maintenance session by focusing their efforts to a few part of the code only (as needed strategy). Alternatively, if time is not limited, or if the intent is to understand all the software details (for migration purpose for instance), the developers will study the code thoroughly so as to systematically discover the purpose and meaning of all the code artefacts. These dual strategies may also be applied to specific part of the software like the components. Indeed, a buggy component may be systematically analysed, whereas the application may only be partially scanned to get the contextual use of the component.

3.5 Integrated Strategy

The integrated strategy considers that the understanding process involve several levels of abstraction simultaneously and opportunistically. For example, each comprehension task may launch sub-processes with the same or another strategy among those identified above. This approach is called an integrated strategy since the tasks and models are jointly participating in a more complex analysis process.

4 UNDERSTANDING MODELS

During the last three decades, the research community proposed several models of the mental processes associated with code understanding. At the same time, the software development and methodologies have deeply evolved as well as the programming languages and environments. In this study we propose to review the most prominent understanding models referred to by the program understanding community since the 70s. In our research, we have identified three major periods whose models targeted different scopes and purposes, generally driven by evolution of the technology:

- The **Classical Period** (before 2000). In this period the software understanding problem was mainly questioned by psychologists interested in software

creation and learning. All the major strategies have been identified during this period;

- The **Optimistic Period** (between 2000 and 2010). In this period, the researchers became interested in the software understanding *process*, with the intent to provide tools and techniques to enhance, if possible automate, software development and maintenance;
- The **Pragmatic Period**, (after 2010). In this period the researchers seemed to return to the genuine program understanding problem with reduced ambitions on the theoretical side. They focus on techniques to ease program maintenance.

5 CLASSICAL PERIOD

This period covers all studies conducted roughly before the year 2000. The five most common strategies involved in software comprehension (Section 3) have been proposed during this very period. Indeed, research was focused on the study of the human behaviour and the associated knowledge models since only limited tools, mainly text editors and debuggers, existed at that time.

5.1 Brooks

Brooks (Brooks, 1983) defines the program comprehension process as the reconstruction of the mappings between the problem domain, possibly through several intermediate levels, and the programming domain (Figure 2). The author argues that this mapping is built iteratively through (1) assumptions made about the program purpose and (2) beacons found in source code or documentation. More precisely, he suggests that the programmer first builds expectations about the program purpose and the corresponding implementation details. From there he identifies which knowledge elements should be included in the mental model to match these expectations. Before inclusion, these hypotheses are validated, refined or rejected against facts found in code or against constraints already present in the current program's model. In other words, the author suggests that the understanding process starts by creating an initial coarse guess about the generic program's purpose through a simple inference made on its name (a beacon). This assumption is then validated against other beacons found in code and accepted, rejected or refined, if relevant. When more beacons are discovered, new intermediate concepts and sub-assumptions are made and the mental model is modified accordingly. The process goes deeper and deeper until the lowest abstraction level is reached, in particular the most technical and code oriented concepts.

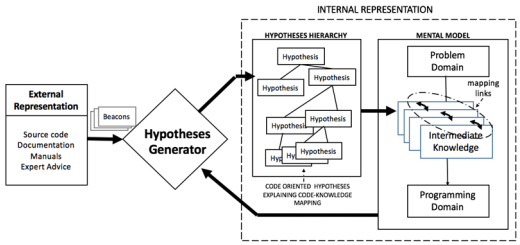

Figure 2: Brooks model, adapted from (Brooks, 1983).

The understanding process is completed when all the beacons found have been matched against multi-level hypotheses, thus explaining the source code through high level description and justification. Since the hypothesis generation process requires a large knowledge base and a wide experience in beacons recognition, this theory suggests that only expert programmers are likely to generate relevant hypotheses and validate them against the proper beacons. In accordance with Exton's classification (§ 3), this model is considered a top-down strategy since high-level hypothesis are generated down to more specific ones, until reaching beacons and implementation structures.

5.2 Soloway and Ehrlich

Soloway and Erhlich (Soloway, 1984) propose to use text comprehension theories to explain how developers understand existing programs from a linguistic point of view. Based on the these theories, they suggest that the programmers use "plan knowledge" and "rules of discourse" (i.e. programming convention) to extract semantic information while analysing source code (Figure 3).

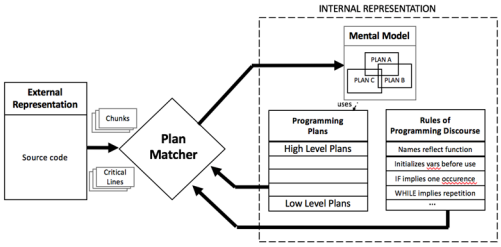

Figure 3: Soloway and Ehrlich model, adapted from (Soloway, 1984).

Plan knowledge, or programming plans, are generic pattern of statements dedicated to reach some specific goal (solve some specific problem). Each plan is labelled with the expected goal. For example, the

search-loop-plan would be associated with the recurrent statements used by programmers to find an item within an array or a collection. So, it will be stored in memory as a strategical plan associated to the *search-loop* label. Thereby, while reading source code, programmers use predefined programming convention and programming plans to shape a compact, goal oriented, program representation in their memory without including too much information details. The authors also found that the plan identification may not use all the statements associated to the plan, but only a part of them called the critical lines. This leads the understanding process to be much faster and efficient. Furthermore, through experiments with novice and expert programmers, the authors also showed that unplan-like programs break programmer expectations and are more difficult to understand for novices. This is true even if the program respects programming conventions. This might explain why the expert programmers are more efficient at identifying functionalities and code purpose, even without reading thoroughly the source code. They can better predict which plans are implemented in the code and what the original programmer intend was. In accordance with Exton's classification (§ 3), this model is considered a top-down strategy, since programming convention and programming plans are pre-defined knowledge that programmers use to recognize code structures.

5.3 Letovsky

Based on experiments conducted with four professional programmers and two novices, Letovsky (Letovsky, 1987) observes that the understanding process can be viewed as an investigation process where the subjects are making small inquiries about the studied program (Figure 4). He suggests that the developers, while analysing code, shape questions about specific fragments of code, make plausible assumptions about them (called conjectures) and evaluate their validity through evidences found in source code. As each conjecture is validated, the programmer's mental model is stepwisely built in memory: new elements are added or removed and relationships between objects are updated. For the author, the mental model of the program is not a uniform network of objects, but rather a layered structure split into three kinds of information:

- The *specification layer*, defining the detailed goals and purposes carried out by the program;
- The *implementation layer*, representing the code structure and statements in subject's memory;
- A set of connections, called *annotations*, linking the specification items to the implementation items. These links "explain" the program's purposes in term of its implementation.

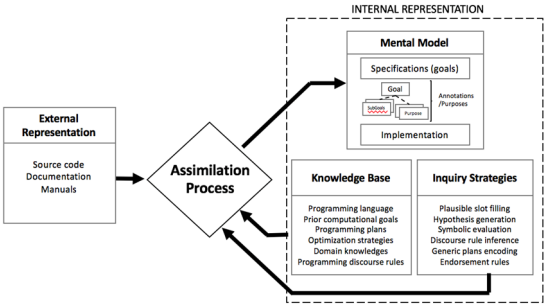

Figure 4: Letovsky model, adapted from (Letovsky, 1987).

The understanding process, viewed as an inquiry process, is called by the author the "assimilation process" (Figure 4). The programmers formulate questions (what, why, how and whether) about the code fragments and make small conjectures about plausible answers. Each question then raises bottom-up or top-down inquiries, and highlights some relationships among concrete and abstract components of the mental model. Then, through annotations, the programmers update their mental model accordingly. When the mental model is incomplete, before the analysed program is entirely understood, the links from goals to implementations may be tangled. Indeed, some goals may be linked to more than one implementation items and some implementation item be associated to multiple goals. Hence the author suggests that the developer, viewed as an opportunistic processor, will use a dual strategy to partially solve conflicting interpretations and delegate full understanding in later stage, when the mental model will have been built more thoroughly. The author acknowledges that many different kinds of knowledge sources are required when doing inquiries and generating plausible assumptions about the code purpose. These are applied in different conditions for different goals: plausibility prediction, hypothesis generation, plan identification, endorsement rules, symbolic simulation and so forth, in order to finally link specifications goals to concrete program implementation. In accordance with Exton's classification (§ 3), this model is considered a hybrid strategy, since a mix of top-down and bottom-up approaches are involved through inquiries.

5.4 Pennington

Also based upon text comprehension theories, Pennington (Pennington, 1987) suggests that the developers use more than one single mental model to

represent knowledge extracted from source code (Figure 5). The key hypothesis is that at least two models are involved in the comprehension process. The first one is dedicated to the textual program representation, the program model, representing the textbase features of the program.

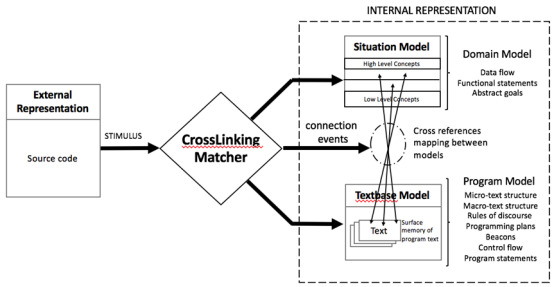

Figure 5: Pennington model, adapted from (Pennington, 1987).

The second one, the situation model, represents what the programmer intended to implement, i.e. the functional features in terms of domain model. The author observes that the programmers, using statements, sequences, data transformation and states, build the program model before the situation model and once done, the latter is shaped by applying proper programming plans justifying the text based structures. Furthermore, the author also suggests that, while these models are being built, specific events, dedicated to crosslinking the elements from both models, are generated. These events, called connection-events, help the developers assign contextual (situation model-based) meaning to code logic represented in the program model. These type of events are triggered sporadically, while the program model is currently being shaped. Surprisingly though, these events seem only to be triggered during code simulation or plan identifica-tion strategies. Experiments clearly showed that if connection-events are missing, or triggered too frequently, the understanding process is deeply disturbed and becomes much less efficient. At the end of an understanding session, the most efficient developers are able to describe program features and functionalities through a balanced set of items distributed among both models. In accordance with Exton's classification (§ 3), this model is considered a bottom-up strategy, since the situation model is built once the textbase model has been identified from source-code.

5.5 Mayrhauser and Vans

Mayrhauser and Vans (Mayrhauser, 1995) propose to merge the Pennington model (§ 5.4) implementing a bottom-up strategy, with the Soloway and Ehrlich model (§ 5.2) implementing a top-down strategy, to form an integrated model using multiple-strategies (Figure 6).

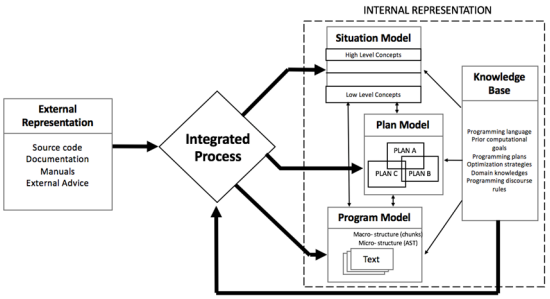

Figure 6: Mayrhauser and Vans model, adapted from (Mayrhauser, 1995).

The resulting model presents three different perspectives or aspects on the same program:

- The text representation through microstructures (abstract syntax tree) and macrostructures (procedural relationship) forming the Program model;
- The functional features based on the domain model (Situation model);
- The multileveled plan hierarchy structure (Plan model).

Each part of the model is associated with a specific process. These processes jointly build the whole mental representations in memory. At any time one process can be suspended and another one may take over to address more specific aspects of the understanding process. Each model has different roles and uses specific information related to the program representation:

- The program model (text-based representation of the code) is build when unfamiliar code is encountered or when a programmer needs to identify precisely what the program is concretely doing;
- The plan model is build when beacons are identified in the code suggesting that a specific function is implemented. This model is goal oriented in the sense that it assigns goals and purpose to the program model elements through meaningful crosslinks between both models;
- The situation model is the most abstract representation of the program. It describes the purpose of the code in functional terms. This knowledge is acquired bottom-up, starting from the program model, or top-down starting from the plan model.

The complete model presented by the authors is the most complex among all models proposed before 2000 and sums up all the strategies into a single and consistent model.

In accordance with Exton's classification (§ 3), this model is considered an integrated strategy, since it involves, at the same time, several understanding approaches.

6 OPTIMISIC PERIOD

This period covers the first decade of the 2000s, object oriented programming became widely used in industry. As this new paradigm changed radically the way software was designed, the research community got more interested in discovering new tools and techniques to help with program understanding and maintenance.

6.1 Rajlich and Wilde

Rajlich and Wilde (Rajlich, 2002) suggests that program comprehension be viewed as a learning process. Their approach is presented through the constructivism theories of learning. Based upon the Piaget's work (Piaget, 1954, Wadsworth, 1996), the authors propose that the assimilation and accommodation theory be applied to the program understanding problem. Indeed, when reading a program, the author argues that developers learn new facts using the Piaget's absorption strategy:

- **Assimilation**: when a new fact fits to the pre-existing knowledge in memory, it is simply added to this knowledge without modifying existing structure;
- **Adaptation**: (called accommodation by Piaget) when a new fact does not fit the pre-existing knowledge in memory, the learner must reorganize his existing knowledge to make it compatible with the new fact.

The authors suggest that the programming knowledge be expressed through the explicit representation of concepts structured as a hierarchical tree (the "conceptual map" presented on the right part of Figure 7). This structure is incrementally built while reading the source code. In this approach, "The Concept Assignment Problem" coined by (Biggerstaff, 1993), i.e. the identification of domain concepts referenced in the code, is a strong prerequisite to code understanding. Then, the authors propose to use the "Software Reconnaissance" techniques, identified by Wilde and Scully (Wilde, 1995), to build a representation of concepts from the code. By analysing runtime traces generated with/ without specific features, the programmers can identify the code involved while activating a specific feature. Therefore, with concepts identification and features reconnaissance, the programmers incrementally build a conceptual map through assimilation and adaptation strategies describing the original program's purpose.

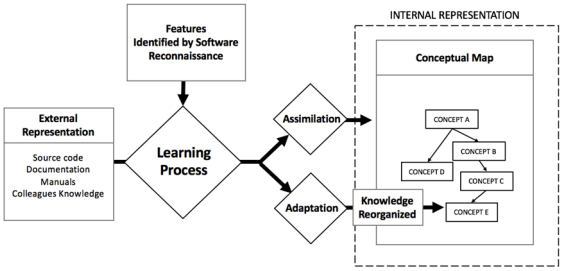

Figure 7: Rajlich and Wilde model, adapted from Rajlich, 2002).

Interestingly, this model uses both a dynamic approach based on the runtime analysis of the execution trace and a static analysis approach with assimilation and accommodation strategies. In accordance with Exton's classification (§ 3), this model is considered a hybrid strategy, since the feature reconnaissance involves a top-down strategy starting from the features and the assimilation-accommodation phase which is a bottom-up strategy.

6.2 Kelson

Kelson (Kelson, 2004) proposes a generic meta-model named EOP (for Event, Operation, Property) to represent all the information gathered during software understanding sessions: the static and the dynamic behaviour of the programs under investigation (Figure 8).

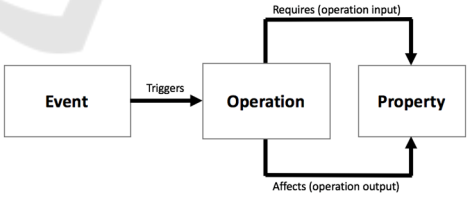

Figure 8: Kelsen meta-model, adapted from (Kelson, 2004).

More specifically, the EOP meta-model is composed of three kinds of elements:

- **Operations**: these represent the actions taken by the system to modify the system's state. They are abstractions of the methods, functions and procedures;
- **Events**: represent the occurrences of data items that activate the operations;
- **Properties**: represent the data items used/ produced by the operations.

To model a specific software's behaviour, the developer needs to instantiate the EOP meta-model to produce a new model, which conforms to UML diagrams, called the EOP-model. The latter describes the dynamic aspects of a specific program in terms of events, operations and properties. Depending on the needed granularity level, the model may also contain several levels of abstraction represented as different layers, with relationships between these layers. In accordance with Exton's classification (§ 3), this model is considered a bottom-up strategy, since all items (operations, events, properties) are first identified manually in the source-code then later abstracted in the EOP-Model.

6.3 Murray and Lethbridge

Based on the design patterns, first introduced by the Gang of Four (Gamma, 1994), Murray and Lethbridge (Murray, 2005) propose to reuse a similar approach in the context of the mental activities involved in software comprehension. First, they defined a *cognitive pattern* as a "structured textual description of a solution to a recurring cognitive problem in a specific context". Like a software design pattern which captures an effective technique for solving a design problem, a cognitive pattern captures a mental operation used by practitioners when trying to understand a program. Generalizing this idea, the authors propose a set of seven high-level cognitive patterns to organize all the mental activities performed during software analysis and investigations (Figure 9). They call it the *micro-theory of understanding*.

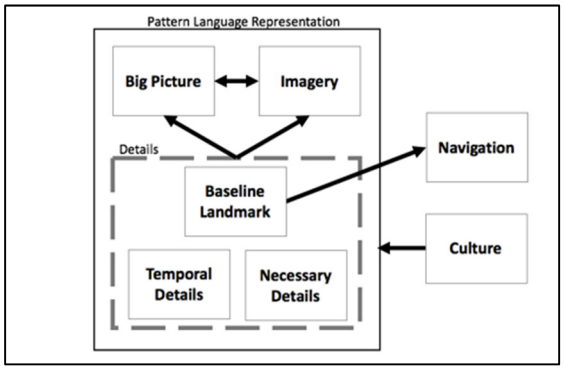

Figure 9: Micro-theory of understanding, adapted from (Murray, 2005).

The arrows in the figure represent the relation "required activity for". The high-level cognitive patterns are themselves composed of more specialized sub-patterns dedicated to specific tasks and purposes. Here are the seven cognitive patterns as identified by the authors:

- **Big Picture:** identifies, at a high level of abstraction, the main purpose of the software;
- **Imagery:** builds a visual representation, through diagrams and symbols, expressing static and dynamic aspects of the software;
- **Baseline Landmark:** identifies the invariants, acquired by experience, allowing the engineers to recognize software components and structures through code navigation;
- **Necessary Detail:** assesses the relevance of a selected strategy (required depth vs inappropriate depth, temporal quality, boundaries and information representation);
- **Temporal Details:** manages the mental models over the time through multiple views and the dynamic aspect of understanding;
- **Navigation:** provides several strategies to analyse code and to build representations.;
- **Culture:** acquires knowledge about software habits, preferred architecture, the constraint and components naming or the documentation techniques.

The authors believe their *Pattern Language* (Alexander, 1977) to provide a handbook of practices and processes which represents common solutions to recurrent problems encountered during understanding sessions. However, the process of software understanding using these patterns is not precisely formalized. Therefore, we cannot assess it using the Exton's classification (§ 3).

6.4 Rilling et al.

To formalize the comprehension tasks and the resources involved while understanding software, Rilling et al. (Rilling, 2006) propose a very high level model of the understanding process (Figure 10).

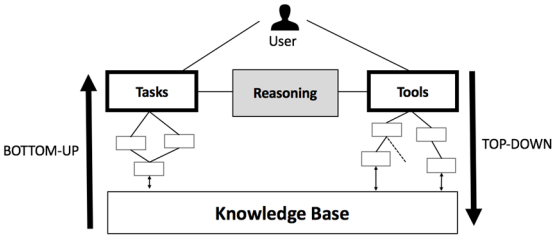

Figure 10: Rilling et al. comprehensive model, adapted from (Rilling, 2006).

The bottom-up process represents the abstraction of high-level information from code. The top-down process consist in mapping this abstract information to concrete code fragments using some understanding tools. The whole process is driven by reasoning rules

on the information already gathered (current state or prior knowledge). However, these rules are not explicit in their paper. Through an ontological model, the authors propose a unique and standard representation of all the tasks and resources used by the developer (Figure 11). Since the model is expressed in OWL-DL (Ontology Web Language Description Logics) it can be queried by the user or by some automated tools. Moreover, the authors describe some scenarios where the elements of this ontology are leveraged during the understanding sessions. This is formalized using a story-metaphor represented as a UML sequence diagram that describes the manipulation of the tasks, artefacts and tools carry out by the developer. This is called the story-manager, which is also represented in the ontological model (Figure 11).

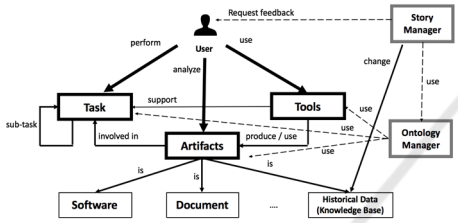

Figure 11: Rilling et al. ontological model, adapted from (Rilling, 2006).

In accordance with Exton's classification (§ 3) this approach is obviously a hybrid strategy.

6.5 Ko Et Al.

Based on a study conducted by Murphy et al. (Murphy, 2005) about the use of some integrated development environment (IDE) when seeking the features of a software, Ko et al. (Ko, 2006) investigate the strategies and practices used by the developers to carry out the maintenance tasks. Applying the *Information Foraging Theory* (Pirolli, 2009), the authors propose a new understanding model based on the strategies performed to maximize the retrieval of valuable information per unit of effort while reading source code (Figure 12).

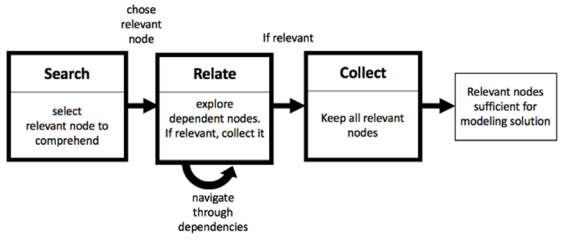

Figure 12: Ko et al. model, adapted from (Ko, 2006).

Similar to an animal's primitive behaviour to retrieve food, the authors suggest that engineers are "foraging" information contained in code and adapt their strategy to minimize the effort spent while retrieving relevant information. Hence they identified a generic search-relate-collect task that processes a graph representing the source-code and its documentation. In this graph, the nodes represent simple pieces of information and the edges the relationship between them (i.e. call, uses, declares, defines, etc.). The generic tasks can be described as follows:

- **Search:** the developer starts by identifying a relevant node in the graph;
- **Relate:** the developer uses clues and documentation to identify all relevant relationship to dependent nodes. For each one, he recursively applies the search and relate tasks again;
- **Collect:** as the search and relate tasks unfolds, the developer gathers the nodes that are necessary for completing the understanding task.

If, at any point in time, the developer believes that the nodes that have been collected are sufficient for the task, the developer quits the process. In the end, the authors suggest that all the collected nodes and their relationships represent the actual mental model persisted within the developer's mind. Fundamentally, this theory proposes a low-level generic understanding process, driven by clues found in the code or the documentation. The authors claim their approach to be compatible with all the classical understanding process presented in section 3. For instance, the hypothesis-based strategy (§ 5.1) and the inquiry-oriented strategy (§ 5.3) correspond to the seeking phase in Ko et al model. Therefore, in accordance with Exton's classification (§ 3) this approach supports them all.

7 PRAGMATIC PERIOD

This period essentially covers research made after 2010, where researchers returned to the core question: what is really code comprehension? Indeed, all the previous works dealt more with the understanding process (how comprehension is reached) than with the comprehension itself. Then, several researchers investigated new measurement techniques, but these works did not lead to new comprehension models. Here are some examples of these works: comprehension measurements (Heitlager, 2007), comprehension simulation (Johnson, 2015), brain activity experiments (Sigmund, 2012 and 2014), and eye-tracking observation (Yusuf, 2007). However other researcher focused on the comprehension itself.

7.1 Belmonte et al.

Belmonte et al. (Belmonte 2014) noted that the kind of information represented in the code is distinct from the information about the purpose or the goal of the program. The high level information describing the program's goal or purpose takes its origin in the business domain while the source-code represent only low-level operations. Hence the source code alone is unable to convey the purpose of the program.

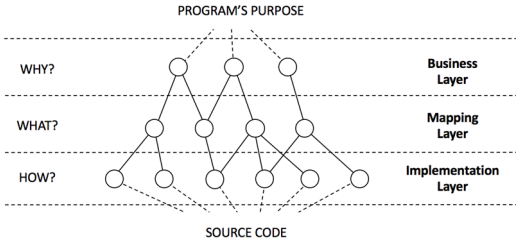

Figure 13: Belmonte et al. model, adapted from (Belmonte 2014).

The authors propose to build an intermediate information layer, the "mapping layer", to link the high-level business functions to the source code methods (Figure 13). Hence, they define the understanding process as the process of reconstructing this intermediate layer. Their model is therefore made of three distinct layers:

- **Top Layer:** describes the program's purpose or functionalities through a sequence of business-oriented information manipulations.
- **Bottom Layer:** low level implementation layer representing the source code methods ;
- **Middle Layer:** mapping layer from the purpose of the program to its implementation. This is considered the "understanding layer".

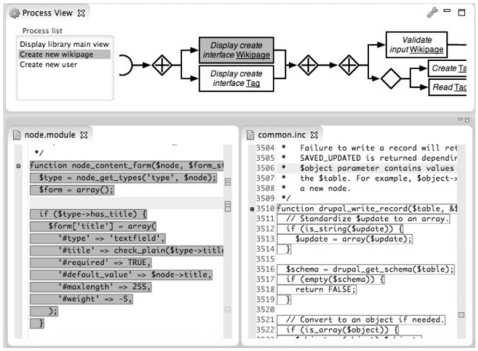

Figure 14: Belmonte et al. example (Belmonte, 2014).

Business-oriented information (top-layer) is represented as a pair of concepts: an action-concept and a domain–concept. This pair represent a single atomic information manipulation task and allows all business workflow to be represented as a sequence of manipulations. Then, the authors propose an original tool to show the potential mappings based on two heuristics: one to retrieve the action-concepts from the code statements and another to identify domain-concepts in the code using a generic pattern matcher on program identifiers. The result is quite elegant, as both business-view and implementation view are display simultaneously (Figure 14). In accordance with Exton's classification (§ 3), this model is considered a top-down strategy, since high level concepts taken from the first layer are used to search the source-code for possible mappings.

7.2 Benomar et al.

Benomar et al. (Benomar, 2015) noted that research on software comprehension was generally split in two distinct areas: program design understanding and program evolution understanding. Hence they propose a unified model encompassing the time-dimension to address both areas: the dynamic aspects of software execution and the evolution over time of the software (Figure 15). In this unified model, the "sequence" is considered the main element for software comprehension. It represents a period of time, with a start time, an end time, and a set of events. An event is an action that occurs periodically in time. It has a time stamp, is triggered by a subject, and has an impact on some objects. Subjects and objects constitute the two entity types involved in the comprehension process. Entities are characterized by properties that are modified by changes introduced over time by the events. This approach allows describing the changes that occur during the execution of the program (during the runtime) as well as the changes to the software itself during its evolution (maintenance).

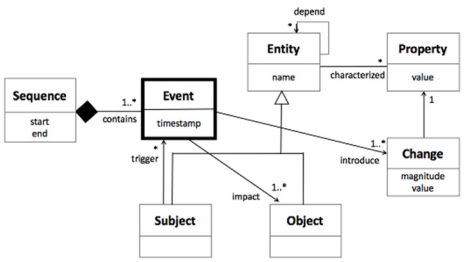

Figure 15: Benomar et al. model, adapted from (Benomar, 2015).

Furthermore, the authors suggest that phases could be identified with respect to the software changes, leading to a partition of the sequence of events into

several sub-sequences. Each phase (sub-sequence of events) must satisfy two specific properties, an internal cohesion property and an external coupling property. Phases are then the main abstraction mechanism through which to understand the software and its evolution, since the authors propose to map high level business events to each of the phases. In addition, the authors also noted that software understanding was drastically impacted by the identification of the collaborations between the entities over time. To illustrate this point, they propose a way to aggregate the entities invocation to later visualize them through a "heat map metaphor". Hence when comparing different maps produced by different execution scenarios, we can clearly observe which classes are really involved in the discriminant feature. Since this model focuses on the low-level description of a running program, it cannot be classified in Exton's classification (§ 3).

7.3 Nosal and et al.

Nosal et al. (Nosal, 2015) observed through controlled experiments that the whole understanding process is hypothesis-based and consists in matching elements found in source-code (solution domain) to the software requirements (problem domain). The authors claim that the understander's current knowledge and prior experiences about the problem and the solution domains is the base on which mapping hypotheses are constructed to recover the mental model used by the original developers. Hence, they proposed to extend the Belmonte et al. model (§ 7.1), with a four-layer model to represent the knowledge involved in the comprehension process (Figure 16):

- The first layer expresses some generic and abstract knowledge about the purpose of the software to understand;
- The second layer is a decomposition of the first layer into atomic features and business concepts;
- The third layer represents beacons and plans that suggest how such features may be implemented;
- The fourth layer consist in a simplified representation of the source code.

The beacons are recognizable static features in code (naming convention, patterns, programming style), while plans are specific algorithms or processes (sorting, parsing, etc.) implemented in code. In the proposed framework, the core comprehension problem is to map beacons/plans to features/concepts in order to link low-level concepts from the solution domain to high-level concepts from the problem domain. Through speak-aloud session, they observed that all engineers started first by playing with the application to catch the general idea and main purpose of the application before delving into the code. These sessions confirmed several assumptions:

- The mapping between the layers is constructed iteratively;
- Software understanding is hypothesis-based;
- If the code is unfamiliar, the hypotheses are based on the interpretation of the identifiers in the program (bottom-up strategy);
- If the code is familiar, a top-down strategy is applied. But the actual technique depends on the engineer's past experience
- If an unexpected identifier is encountered this triggers a systematic analysis code details, otherwise the as-needed strategy is generally preferred.

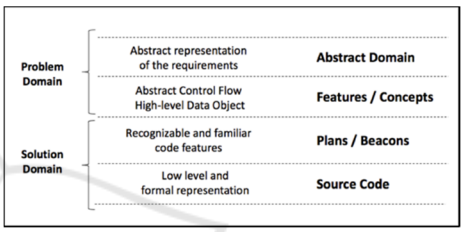

Figure 16: Nosal and Poruban model, adapted from (Nosal, 2015).

The model proposed by the authors is quite simple and practical. It also seems compatible with most of the software understanding process previously identified by researchers. In accordance with Exton's classification (§ 3), this model allows all strategies to be potentially useable.

8 DISCUSSION

This research work is based on the papers addressing only understanding models. From the 155 papers we analysed which referenced such models, we noted that 37% were published before 2000, 43% between 2000 and 2010, and only 20% after 2010. This clearly indicates that the interest in modelling understanding reached a peak at the beginning of the 2000s. Besides, it also shows that most studies related to software understanding models closely followed the evolution of the techniques and tools in software design. Before the year 2000, most of the tools used by practitioners were text-oriented (text editors, text debuggers and text interfaces). However, in the first decade of 2000, much richer and interactive environments (IDE), graphical debuggers and visual editors were adopted. From the mid 90's, the OO paradigm, pioneered by

Simula 67 (Hills, 1972) and Smalltalk 80 (Goldberg, 1983), finally reached the mainstream development market through the C++ (Stroustrup, 1997) and Java (Gosling, 1996) languages ecosystems. These technological trends had obviously a strong influence on the research works on software understanding processes. We found that most of the models addressed the techniques and strategies used by the developers when analysing software. In other words, they focus more on the *process* of understanding: what the developers do to understand a program. But very few papers addressed the more fundamental question: "what does it really mean to understand a program"? This distinction was already made by Storey (Storey, 2005) several years ago. Some people may claim that "everyone knows what understanding mean". But we do not share this statement. We believe that the actual task of "understanding" a piece of code has not been studied enough to let engineers design useful tools. An informal proof of this is the well documented finding that most of the tools intended to "help with program understanding" are generally underutilized by developers (Lanza, 2003) (LaToza, 2006) (Pacionne, 2004) (Roehm, 2012). In our survey we observed the following facts. During the classical period the authors proposed internal models (i.e. in the mind) of the knowledge required for a developer to understand code, and the process to build them. During the optimistic period that followed, the authors addressed the understanding process from which they propose tools to offer alternative representations of source code. The latter were based on some visual metaphor and aimed at helping developers to navigate the code. During the last decade, the research community became less active on the topic of code understanding models. This may be due to the slow progress made toward the fundamental goal. However, a few researchers started back from the key question: what is really understanding code? The answer they proposed is based on the idea of mapping representations of the business concepts to the code through different levels of abstraction. But such a mapping is not straightforward. Belmonte et al. (Belmonte, 2014) proposed a three layers' model while the one of Nosal et al. (Nosal, 2015) rested on four layers. Yet, the latter can be considered as complementary to all the understanding process models that have been proposed during the classical period.

9 CONCLUSION

This paper presents a review of the last 30 years of publications on the topic of program understanding models. We proposed a classification of the works in three periods, since we found the papers to address three different perspectives on the topic along time: the process, the tools and the goal. The general conclusion is that the fundamental question "what is code understanding" was only recently addressed by the research community. This is surprising since tools have been developed without a clear view on what "understanding code" means. Another finding is that the number of papers on the topic decreased radically during the last decade. We suppose this to be due to the lack of progress toward the fundamental question. For example, the papers in the early 2000s focused on the development of program understandings tools. But these tools are showed to be largely underutilized by the developers. We suggest that the fundamental question must still be thoroughly investigated before going any further in the implementation of tools. In fact, our team is now working on this very question since we believe the code understanding problem (i.e. its cost) to be ever more acute today.

REFERENCES

Alexander, C., 1977. *A Pattern Language, Towns, Buildings, Construction*, New York, Oxford University Press.

Belmonte C., Dugerdil, A., Agrawal, A., 2014, *A Three-Layer Model of Source Code Comprehension*, Proceedings of the 7th India Software Engineering Conference, Article no. 10.

Benomar, O., Sahraoui, H. Poulin, P., 2015. *A unified framework for the comprehension of software's time dimension*, 37th International Conference on Software Engineering vol. 2, pp. 603-606.

Biegel, B., Baltes, S., Poulin, P., Scarpellini , I., Diehl, S., 2015. *CodeBasket: Making Developers'Mental Model Visible and Explorable*, IEEE/ACM 2nd IWCSD 2015.

Biggerstaff, T.J., Mitbander, D.G., 1993, *The Concept Assignment Problem in Program Understanding*, ICSE '93 Proceedings of the 15th International conference on Software Engineering, pp 482-498.

Brooks, R., 1983, *Towards a theory of the comprehension of computer programs*, International Journal of Man-Machine Studies, vol. 18, no. 5, pp. 543-554.

Exton, C., 2002, *Constructivism and Program Comprehension Strategies*, IWPC'02 Proceedings of the 10th International Workshop on Program Comprehension, p. 281.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series.

Goldberg, A., Robson, D., 1983, *Smalltalk 80, the language and its implementation*, Addison-Wesley Publishing Company, Xerox Palo Alto Research Center.

Gosling, J., Joy, B., Steele, G., 1996, *The Java Language Specification*, Addison-Wesley Publishing Company.

Hein, G. E., 1991, *Constructivist Learning Theory*, CECA

Conference, Jerusalem Israel, 15-22 Oct 1991.

Heitlager, I., Kuipers, T., Visser, J., 2007, *A Practical Model for Measuring Maintainability*, QUATIC'07, 6th International Conference on Quality of Information and Communications Technology, pp. 30-39.

Hills, R., 1972, *Simula 67, an introduction*, Robin Hills Ltd, Presto Print, Reading.

Johnson, P., Ekstedt, M., 2015, *Exploring Theory of Cognition for General Theory of Software Engineering*, 4th SEMAT Workshop on a General Theory of Software Engineering.

Kelson, P., 2004, *A Simple Static Model for Understanding the Dynamic Behavior of Programs*, 12th IEEE International Workshop on Program Comprehension (IWPC'04).

Ko, A.J., Myers, B.A., Coblenz, M.J., 2006, *An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks*, IEEE Transactions on Software Engineering, vol. 32, no. 12.

Lanza, M., Ducasse, S., 2003, *Polymetric Views – A light-weight Visual Approach to Reverse Engineering*, IEEE Transactions on Software Engineering, Vol. 29, p. 9.

LaToza, T.D., Venolia, G., DeLine, R., 2006, *Maintaining Mental Models: A Study of Developer Work Habits*, ICSE'06, Shanghai, China.

Letovsky, S., 1987, *Cognitive process in program comprehension*, Journal of Systems and Software, vol. 7, no. 4, pp. 325-339.

Mayrhauser, von A., Vans, A.-M., 1995, *Program Comprehension During Software Maintenance and Evolution*, IEEE Journal Computer, vol. 28, no. 8, pp. 44-55.

Muchalintamolee, V., 2012, *Measuring Granularity of Web Services with Semantic Annotation*, AIJSTPME, vol. 5, no. 3, pp. 41-48.

Murphy, G.C., Kersten, M., Robillard, M.P., Cubranic, D., 2005, *The Emergent Structure of Development Tasks*, ECOOP'05 Proceedings of the 19th European conference on OOP, pp 33-48.

Murray, A., Lethbridge, T.C., 2005, *Presenting Micro-Theories of Program Comprehension in Pattern Form*, ECOOP'05 Program Comprehension, IWPC 2005, 13th International Workshop on Program Comprehension.

Nosal, M., Poruban, J., 2015, *Program Comprehension with Four-layered Mental Model*, 13th International Conference on Engineering of Modern Electric Systems.

Pacione, M.J., Roper, M. Wood, M., 2004, *A Novel Software Visualisation Model to Support Software Comprehension*, 11th Working Conference on Reverse Engineering 2004.

Pennington, N., 1987, *Empirical studies of programmers: second workshop*, Ablex Publishing Corp. Norwood, NJ, pp. 100-113.

Pennington, N., 1987, *Stimulus structures and mental representations in expert comprehension of computer programs*, Cognitive Psychology, vol. 19, no. 3, pp. 295-341.

Piaget, J., 1996, *Piaget's theory of cognitive and affective development: Foundations of constructivism*, 5th reedit. Wadsworth, Barry J., White Plains, NY, England.

Piaget, J., 1954, *The construction of reality in the child: Foundations of constructivism*, Basic Books, New York.

Pirolli, P., 2009, *An Elementary Social Information Foraging Model*, CHI 2009, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 605-614.

Rajlich V., 2002, *Program Comprehension as a Learning Process*, IEEE International Conference on Cognitive Informatics, p. 369.

Rajlich V., Wilde, N., 2002, *The Role of Concepts in Program Comprehension*, 10th International Workshop on Program Comprehension 2002, pp. 271 - 278.

Rilling, J., Mudur, S., Charland, P., Witte, R., Meng, W.J., Zhang, Y., 2006, *A Context-Driven Software Comprehension Process Model*, IEEE International Workshop on Software Evolvability, pp. 50-55.

Roehm, T., Tiarks, R., Koschke, R., Maalej, W., 2012, *How Do Professionnal Developers Comprehend Software?*, ICSE 2012, Zürich, Switzerland, 2012.

Shneiderman, B. Mayer, R., 1979, *Syntactic/semantic interactions in programmer behavior: A model and experimental results*, Journal of Computer and Information Science, vol. 8, pp. 219.

Sigmund, J., 2012, *Framework for Measuring Program Comprehension*, PHD Dissertation, Otto-von-Guericke-Universität Magdeburg.

Sigmund, J., Kastner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., Saake, G., Brechmann, A., 2014, *Understanding Understanding Source Code with Functional Magnetic Resonance Imaging*, ICSE'14, Hyderabad, India.

Smith, D., Thomas, B., Tilley, S., 2001, *Documentation for Software Engineers: What is Needed to Aid System Understanding*, SIGDOC 2001, Santa Fe, New Mexico, USA.

Soloway, E., Ehrlich, K., 1984, *Empirical Studies of Programming Knowledge,* IEEE Transactions on Software Engineering, vol. 10, no. 5, pp. 595-609.

Storey, M.A., 2005, *Theories methods and tools in program comprehension: past present and future,* IWPC 2005, pp. 181-191.

Stroustrup, B., 1997, *The C++ Programming Language,* ATandT Labs, Addison-Wesley.

Tilley, S.R., Smith, D.B., 1996, *Coming Attractions in Program Understanding,* Software Engineering Institute, Carnegie Mellon University.

Wadsworth, B.J., 1996, *Piaget's theory of cognitive and affective development: Foundations of constructivism,* 5th ed. xi, 1996.

Walenstein, A., 2002, *Theory-based Analysis of Cognitive Support in Software Comprehension Tools,* IWPC'02 IEEE.

Warintarawj, P., Laurent, A., Huchard, M., Lafourcade, M. Pompidor, P., 2013, *Software understanding: automatic classification of software identifiers,* Intelligent Data Analysis, IOS Press, vol 19, no 4, pp. 761-778.

Wilde, N., Scully, M.C., 1995, *Software reconnaissance: mapping program features to code,* Journal of Software Maintenance, vol. 7, no. 1.

Yusuf, S., Kagdi, H., Maletic, J. 2007, *Assessing the Comprehension of UML Class Diagrams via Eye Tracking,* 15th IEEE International Conference on Program Comprehension.