Heinonen, Ava; Lehtelä, Bettina; Hellas, Arto; Fagerholm, Fabian

Synthesizing research on programmers’ mental models of programs, tasks and concepts — A systematic literature review

Published in:
Information and Software Technology

DOI:
10.1016/j.infsof.2023.107300

Published: 01/12/2023

Document Version
Publisher's PDF, also known as Version of record

Published under the following license:
CC BY

Please cite the original version:

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.
Synthesizing research on programmers’ mental models of programs, tasks and concepts — A systematic literature review

Ava Heinonen *, Bettina Lehtelä, Arto Hellas, Fabian Fagerholm

Department of Computer Science, Aalto University, P.O. Box 15400, FI 00076 AALTO, Helsinki, Finland

ARTICLE INFO

Keywords:
Mental model
Mental representation
Human factors
Program comprehension
Psychology of programming
Programmer
Software developer
Software development
Systematic literature review
Empirical software engineering

ABSTRACT

Context: Programmers’ mental models represent their knowledge and understanding of programs, programming concepts, and programming in general. They guide programmers’ work and influence their task performance. Understanding mental models is important for designing work systems and practices that support programmers.

Objective: Although the importance of programmers’ mental models is widely acknowledged, research on mental models has decreased over the years. The results are scattered and do not take into account recent developments in software engineering. In this article, we analyze the state of research on programmers’ mental models and provide an overview of existing research. We connect results on mental models from different strands of research to form a more unified knowledge base on the topic.

Method: We conducted a systematic literature review on programmers’ mental models. We analyzed literature addressing mental models in different contexts, including mental models of programs, programming tasks, and programming concepts. Using nine search engines, we found 3678 articles (excluding duplicates). Of these, 84 were selected for further analysis. Using the snowballing technique, starting from these 84, we obtained a final result set containing 187 articles.

Results: We show that the literature shares a kernel of shared understanding of mental models. By collating and connecting results on mental models from different fields of research, we provide a comprehensive synthesis of results related to programmers’ mental models.

Conclusion: The research field on programmers’ mental models faces many challenges arising from a lack of a shared knowledge base and poorly defined constructs. By creating a unified knowledge base on the topic, this work provides a basis for future work on mental models. We also point to directions for future studies. In particular, we call for studies that examine programmers working with modern practices and tools.

1. Introduction

It is generally accepted that programmers form mental models of programs, tasks, and concepts [1–3]. The term mental model denotes a mental representation of an external entity. People understand external reality by developing these mental models that are then used to understand the world, reason about it, predict the outcomes of situations and actions, and make decisions [4–6]. The theory of mental models has allowed researchers to explain how programmers perform complex cognitive tasks. This includes comprehending source code to form an understanding of a program that encompasses not only the source code itself but also the abstract semantic aspects of the program [7], and how programmers can form an understanding of code that should be implemented to meet task requirements [8]. It also includes how programmers store and utilize knowledge of abstract programming concepts. This knowledge encompasses semantic knowledge of the concept, syntactic knowledge of its implementation in different programming languages, and how the code implementing the concept behaves at runtime [2].

Supporting programmers in developing accurate mental models remains an important goal and an active research topic in program comprehension [9], computing education [10], empirical software engineering [11] and tool development [12]. Different tools and techniques for supporting programmers in developing accurate mental models have been developed [9,12,13], yet their efficacy and adoption into wider use has been lacking so far [9,13,14]. This has led to
calls for further research into programmers' mental models and their
development [1,13].

However, existing research on programmers' mental models is scat-
tered, and fragmentation of the knowledge base has resulted in a
situation in which existing results are difficult to connect into a co-
herent whole [1]. This has led to a lack of shared language [15] and
scattered research efforts [1]. Therefore, many studies do not build on
existing results [1,16,17], thus hindering the progress of the research
field.

In this article, we present the results of a systematic literature
review of 187 articles on programmers' mental models from different
research fields. This study aims to bring coherence into the scattered
research field by analyzing the research that has been conducted so
far, assessing how mental models are conceptualized and described
in the literature, and bringing together results from across computing
literature to form a coherent synthesis of the main results related to
programmers' mental models.

Our study addresses the following research questions:

RQ 1. How have programmers' mental models been studied?
RQ 2. How have programmers' mental models been conceptualized
and described?
RQ 3. What do the main results in the literature indicate about differ-
ent aspects of programmers' mental models?

With the first research question, our objective is to analyze existing
studies focusing on the target systems, participant groups, study tasks,
and study environments that have been used in the studies. This article
contributes a comprehensive overview of the existing literature on pro-
grammers' mental models and highlights gaps in the existing research
coverage, forming avenues for further research.

With the second research question, we aim to evaluate how the
research field understands mental models by analyzing how mental
models are conceptualized and described. As yet, we do not have
generally accepted definitions of programmers' mental models. This
article contributes an analysis of how mental models are conceptualized
and described across the research field. We show a kernel of shared
understanding across the research field and discuss how it relates to
other fields of research.

With the third research question, summarize existing knowledge on
programmers' mental models by identifying results related to the same
aspects of mental models from across the research field and providing
a synthesis of these results. This article demonstrates that it is possible
to unify the scattered knowledge that exists in the field and contributes
a coherent description of a common kernel of the existing knowledge
base, which unifies the current knowledge and provides a more solid
base for further inquiry into programmers' mental models.

This article is organized as follows. In Section 2, we describe the
background of the study. In Section 3, we detail our systematic
literature review was conducted. In Sections 4, 5, and 6, we present
the results of the review. In Section 7, we further discuss the results in
relation to our research questions. We also discuss the validity of the
study. Finally, in Section 8, we present concluding remarks and provide
directions for future research.

2. Background

2.1. Mental models

One of the first researchers to discuss Mental Models was Craik,
who proposed that people build “small-scale models of the world” in
their minds, and that these models allow them to mentally simulate
different courses of action to select the correct course of action in any
one situation [18]. Since then, this idea of internal representations
that people can use to understand the world, predict outcomes, reason,
and make decisions has been used to explain many aspects of human
knowledge and cognitive performance, including how people form an
understanding of the meaning of a text, the form of knowledge stored
in memory, and user’s understanding of mechanical machines and
different systems and processes [5,6].

Research into the exact neural mechanisms of mental models and
their development in the human brain is ongoing [19]. However, the
core notion that humans understand the world and different entities
within it by modeling it in their mind is widely accepted and supported
by existing research efforts [4,19,20].

2.1.1. Describing mental models

There has been substantial interest in creating ways to repre-
sent mental models as diagrams or other external representations.
Researchers have created different representations of programmers' 
mental models and the program comprehension process. For example,
Von Mayrhauser details some existing ways in which prior research
has done so [21]. Other representations have also been presented, for
example, by Schauer [22] and Frey [23]. However, it appears that many
of the existing representation approaches have not been adopted for
wider use in further research efforts.

Researchers have also made efforts to represent individuals’ men-
tal models to communicate their information content. These external
representations of mental models are called conceptual models [24].
Conceptual models have been used to communicate the information
content of expert mental models of different entities such as VPNs [25],
or adversarial machine learning [26].

In this paper, we will focus on mental models rather than their
conceptual representations. In other words, we focus on the mental
models themselves rather than the means of making them visible
outside a person’s mind. Representing a person’s mental models as a
conceptual model can be beneficial for research and communication
purposes, while the focus of this paper is to work towards providing
a knowledge base on programmers’ mental models that would aid in
these efforts.

2.1.2. Mental models in computer science literature

In computer science, mental models have been used to explain
programmers’ understanding of programs [4,27], programming-related
tasks [3,28] and concepts [2].

Programmers can develop an understanding of a program from
its source code that encompasses not only an understanding of the
meaning of the code statements, but also a more abstract understanding
of the functional entities formed by the code statements, and how these
entities work together to form the functionality of the program [29].
In program comprehension research, the programmers’ understanding
of the program is referred to as their mental model [21].

Different theories from other fields of research have been utilized
to understand program comprehension and the resulting mental mod-
els [7]. Theories based on text comprehension posit that program
comprehension is similar to text comprehension, and thus text compre-
hension theories can be applied to code comprehension situations [7,
30]. Theories based on schema theory approach program comprehen-
sion as a process of activating schemata, or knowledge structures stored
in long-term memory [31].

The different theoretical approaches agree that the end result of
program comprehension is a coherent mental model of the program.
The mental model is often understood as a working-memory construct
that aids in working with the program code [20]. Common to text-
comprehension-based theories is the idea of a layered mental model.
This layered mental model consists of representations of the code at
different levels of abstraction—understanding of the code syntax and
understanding of the semantic meaning of the code syntax are stored
as separate yet interconnected layers of a mental model [7]. Schema-
based theories provide a less detailed description of the structure of
the mental model. According to schema-based theories, the result-
ing mental model consists of some connected collection of activated
schemata [3,32,33].
The proposed theories differ in their understanding of program comprehension or in how programmers acquire their mental models of programs. Pennington hypothesized that programmers would comprehend code bottom-up, or by making inferences from reading the source code and using knowledge of the programming language to understand the code statements [30]. Other theorists proposed that program comprehension occurs top-down, or by recognizing familiar code elements and building a high-level understanding first before delving into implementation-level details [7,34]. For example, schema-based theories highlight top-down processes in program comprehension and the role and use of hypotheses in program comprehension as expectations or inferences created from schema activation [7]. Although both theories show some experimental backing, when studying programmers comprehending large-scale programs, von Mayrhauser et al. deduced that programmers use both top-down and bottom-up processes during program comprehension, switching between them as required [35-37]. Her integrated metamodel theory thus combines top-down and bottom-up theories [38].

Programmers’ understanding of a task or a solution is also called their mental model of the task or the solution. However, studies differ in their definitions of what a task mental model represents. Understanding an existing program that is being modified, the required modifications, and the changes necessary to modify the program. Studies have proposed different ideas on whether all these aspects of a task form one mental model or if they are multiple separate mental models [3,39,40].

In education contexts, understanding of mental models and their development is derived from the theory of model-centered learning and understanding of learning as the process of constructing mental models [41]. These theories have been used in computer science research, where the students’ understanding of programming concepts is referred to as their mental models of these concepts. These mental models are understood as information stored in long-term memory. Studies have classified students’ mental models of concepts such as recursion. They have described these mental models as the student’s understanding of the concept. This includes understanding how a piece of code implementing the concept behaves or what happens at runtime when code implementing the concept is run [10,45,46].

2.2. Related reviews

Researchers have examined the concept of mental models and evaluated the research field in previous publications.

Staggers and Norcio [47] explored mental models in human–computer interaction. They analyzed existing theories of mental models and explained what mental models mean in the context of human–computer interaction. However, the review is thirty years old, and the research field has evolved. In this study, we have included the last thirty years of research to update the picture of mental models.

In recent years, two literature reviews have been published to organize the scattered research field. These studies show the raising interest and importance of this field and the interest in organizing the research field to allow for further research that builds on the existing theories and knowledge. One recent review provided a synthesis of existing code comprehension theories [48]. A recent systematic literature review analyzed 71 studies to evaluate existing research on programmers’ mental representations of code in programming situations [1]. In our study, we have included research into programmers’ mental models of tasks and concepts to provide a more comprehensive understanding of the topic and to provide a more detailed synthesis of the existing results on programmers’ mental models.

Studies in other disciplines have also analyzed mental models. For example, Rook [49] assessed the concept of mental models to build a definition of mental models for organizational management. Jones et al. [6] completed a similar study that analyzed research on mental models from multiple disciplines to provide the field of natural resource management with a unified model of mental models.

3. Research method

3.1. Search method

3.1.1. Pilot search

In the pilot search stage, we created a pilot set of articles. The pilot set was used to create the search string and select article databases. To create the pilot set, we selected three articles on programmers’ mental models [34,50,51] and a recent systematic literature review [1] as starting points. We did a step of forward and backward snowballing, which produced 75 articles. We then performed manual searches using the keywords identified so far in the process, which yielded 22 articles. The results were evaluated on the basis of their title and abstract. After evaluation, the pilot set contained 52 articles.

3.1.2. Search string

We extracted relevant terms and synonyms for target populations, mental models, and target systems from items in the pilot set. To form the search string, we combined terms for (1) the target population, (2) mental models, and (3) relevant target systems. We tested multiple iterations of the search string using different terms and synonyms. We evaluated the number of results, the number of relevant results, and the degree to which searches were able to find articles in the pilot set. We used two versions of the search string. In databases that support either the W/n operator or the synonymous NEAR/n operator, we used the following search string:

1. ( programmer OR coder OR developer )
2. AND ( mental OR "mental model" OR "mental models" OR "mental representation" OR "mental representations")
3. AND ( code OR program OR software OR compute )

In databases that did not support these operators, we used the following search string:

1. ( programmer OR coder OR developer )
2. AND ( mental OR "mental model" OR "mental models" OR "mental representation" OR "mental representations")
3. AND ( code OR program OR software OR compute )

In databases that supported limiting the search to title, abstract, and keywords, this filtering was used. In databases that did not allow for this filtering, the search was performed on the fields available in those databases.

3.1.3. Database selection

We used the items in the pilot set to identify and select relevant databases. The nine selected databases and the number of results from each database are described in Table 1. We performed the queries on 30 June and 1 July 2020.

3.2. Inclusion and exclusion criteria

To provide a comprehensive analysis of the research field, we decided not to filter studies by their publication date. Although many of the included studies were conducted using technologies that are no longer popular, the early studies developed theories of mental models and their acquisition that are still in use today. Therefore, leaving them out of our result set would have limited our synthesis of results. We used the following two inclusion criteria:

IC 1. Studies whose main focus is on studying the structure or acquisition of programmers’ or programming students’ mental models of different target systems or programming activities.
We also defined nine exclusion criteria:

**EX 1.** Papers on shared mental models or group mental models.

**EX 2.** Papers on specific brain patterns or brain activity when no link to mental models is demonstrated.

**EX 3.** Papers on technological interventions or tools developed to display, describe, or communicate mental models if the article does not contain research and results related to mental models themselves.

**EX 4.** Papers concerning the development of artificial mental models for artificial intelligence or robots.

**EX 5.** Papers concerning users’ mental models or end-user development, except for developers or software development students using an end-user-programming system for software development activities.

**EX 6.** Papers on non-developers’ mental models of target systems or development activities.

**EX 7.** Papers not written in English.

**EX 8.** Short papers, such as extended abstracts or idea papers.

**EX 9.** Non-academic sources where the quality of the information cannot be determined.

### 3.3. Study filtering process

The set of results from the database searches was filtered to exclude irrelevant results.

#### 3.3.1. Duplicate removal and initial filtering

First, the results were combined to form the initial result set. Then an automated duplicate check was performed using the Mendeley Desktop duplicate check tool. Duplicates detected by the tool were verified and merged using the Mendeley Desktop tool. This removed 605 duplicate results, leaving 3678 results.

Duplicate removal was followed by initial filtering. The items in the initial result set were evaluated based on their type, language, and title. This was performed by one researcher and removed 2916 items, leaving 762 items in the initial result set.

#### 3.3.2. Exclusion rounds

Initial filtering was followed by two exclusion rounds, E1 and E2. During E1 and E2, the initial result set was divided between the research team to evaluate and decide on inclusion or exclusion. A second researcher verified all exclusion decisions. If the researchers disagreed, they discussed the decision to reach agreement, consulting the other researchers if necessary. During E1, items were evaluated based on their title. During E2, they were evaluated based on title and abstract. The two exclusion rounds excluded 566 items, leaving 196 items in the initial result set.

During data extraction, another 84 items were excluded, leaving 187 results in the initial result set. Fig. 1 shows the study filtering process and details the number of items included in each step of the process.

### 3.4. Snowballing

To achieve better coverage, we performed one round of forward and backward snowballing, following Mourao et al. [52]. In backward snowballing, the reference list of each included item was evaluated, and relevant references were included. In forward snowballing, Scopus and Google Scholar were used to find publications that cite each included item. The publications were evaluated based on their title and abstract, and the relevant publications were included. After removing duplicate results, we had 1564 items in the snowballing set. It was then filtered using the process described in 3.3. After filtering, the snowballing set contained 346 items. During data extraction, 190 items were excluded from the snowballing set. The final result set contains both the initial result set and the snowballing set. A further 53 items were excluded during data analysis. The final result set therefore contains 187 results.

### 3.5. Data extraction

We designed a data extraction form to systematically extract data from the results. A summary of the data extraction form is provided in Table 2. For details of the data extraction form, see the supplementary material in Appendix A.

The data extraction form was created by the first author. An evaluation meeting was held with all authors to assess the form. The form was further evaluated and adjusted before snowballing after data was extracted from items in the initial result set.

To calibrate consensus between researchers, a series of trial extractions were performed. All authors extracted data from the same articles. A series of meetings was held to discuss extraction. These meetings were held until no significant differences in interpretation emerged anymore.

At first, each author extracted data in sets of six items. During snowballing, the set size was increased to 10. If the extractor had any doubts about a paper, they were discussed in a meeting. From each set, another researcher verified one data extraction and all exclusions. Any disagreements were resolved by discussion between the extractor and the verifier.

### 3.6. Data analysis

We relied on frequency analysis and thematic analysis to analyze the data and answer our research questions.

#### 3.6.1. Study classification

For study details, we classified the studies into categories that were determined based on the analysis of the pilot set and discussions during the evaluation of the data extraction form. The categories are shown in Table 2. Studies were assigned to these categories during data extraction. The frequency of these was visualized to answer RQ1.
3.6.2. Thematic analysis

For the analysis of the mental model descriptions, a modified version of the process recommended by Cruzes and Dybå [53] was used. An initial codebook was created by inductive coding of the initial result set. Coding was used to identify the main concepts and themes in the data. Initial coding was carried out by two researchers independently. The final codebook was then created through discussion between them. The final codebook was used to code the snowballing result set.

The results related to mental models were also synthesized using thematic analysis, following the practices recommended by Guest et al. [54]. Text excerpts related to the results were first coded to identify common aspects of mental models that have been studied. Results related to each identified aspect were then coded and synthesized. Coding was carried out by one researcher. The analysis was then evaluated and verified by the entire research team.

3.7. Overview of the result set

Our final result set contained 187 results published between 1977 and 2020. Items included in the final result set are listed in Appendix B. The result set contained 88 journal articles, 59 conference papers, 23 workshop or meeting publications, 14 chapters in edited collections or academic books, 2 technical reports, and one pre-print article. Of the 187 results, 166 reported a participant study. The distribution of the articles over decades is shown in Fig. 2. A total of six articles from 2020 met our inclusion criteria. However, we ran the searches in July 2020. This leaves out some publications published later in the year.

4. How have mental models been studied?

4.1. Target systems and participants

Fig. 3 shows the number of articles in each target system category and subcategory. We describe these categories and subcategories in Table 2.

We also analyzed the programming languages that were used in the studies. The most commonly used programming languages were Pascal (21 results), Java (16 results), Fortran (10 results), C++ (9 results), and BASIC (9 results).

Our results show an almost even split between novice and expert participants, with novice and expert referring to the categorizations given in the articles themselves. However, most of the participants have been students. Fig. 4 shows the number of items in each participant level and participant type category. We describe the categories in Table 2.

4.2. Study contexts and study formats

Most of the studies were conducted in laboratory settings and used paper- or computer-based tests we categorized as non-programming tasks. The number of studies in each study context and study task category is shown in Fig. 5. We describe the categories used to classify this data in Table 2.

Non-programming tasks include study tasks that did not involve coding in a realistic computing setting but were conducted in an artificial setting, where program code could not be executed. For example, the task could involve program comprehension tasks based on source code snippets printed on paper.
4.3. Theories

We identified a set of reoccurring theoretical concepts and analyzed the number of results that mention each concept. We show the theoretical concepts in Table 2.

Several studies did not mention any theoretical concepts. Furthermore, the spread of different theories within this field is wide. Many studies mentioned concepts that we did not see in other studies. These concepts were assigned to the “other” category. The number of studies that mention each theoretical concept is detailed in Fig. 6.

5. How have programmers’ mental models been conceptualized and described?

Our analysis revealed three themes in how the studies described mental models: (1) overview of mental models (2) defining characteristics of mental models, and (3) mental model acquisition. Within these themes,
we identified different aspects of mental models and their acquisition. The aspects contained within each of the themes are described in Table 3.

5.1. Overview of mental models

5.1.1. Mental model definitions
Studies defined mental models in two ways: as cognitive structures or as conceptualizations of thought. Cognitive structures include definitions such as internal models [55] and internal representations [42, 56–61]. Conceptualizations of thought include definitions such as the abstraction of knowledge [62] and the characterization of understanding [63–67].

Some studies did not take a position on areas of memory associated with mental models. Some studies describe them as related to short-term memory or working memory [36,57,68–70]. Long-term memory as the location of the programmers’ knowledge base is mentioned in the context of, e.g., schema-based theories [31].

5.1.2. Types of target systems
The most common target systems were systems or programs including various aspects of a program or a system, such as its source code or the underlying machine [11,30,31,33,37,55,58–60,71–79].

Target systems also included programming concepts [2,42,43,56,66, 67,80–82], and programming situations. Programming situations include
programming tasks and solutions [3, 40, 40, 70, 83–86], and programs being implemented or designed [8, 85, 87, 88].

5.1.3. Uses and influence

Mental models were described as helping perform tasks such as decision-making [28, 61], making predictions [8, 28, 50, 66, 89], and programming tasks [69, 83, 90] such as debugging [55, 61] and program modification [55, 77, 91]. Some articles also mention that mental models influence task performance. They mention that mental models help perform tasks, for example, by reducing the cognitive load during program comprehension [16].

More detailed descriptions of the uses of mental models were also present. Mental Models were described as mental representations that allow one to derive information about the target system through mental simulation [50, 90, 92] and reasoning [8, 56, 93]. Some studies described them as representations that, in general, help interpret and explain the target system [3, 28, 61, 89, 93].

5.2. Defining characteristics

5.2.1. Information content

Some articles described mental models as containing knowledge and information related to the target system [31, 31, 55, 59, 60, 76, 84, 93–95]. Others include strategies to comprehend the system and background knowledge related to the system and the situational context as parts of the mental model [93].

Some articles specify that the knowledge in mental models represents either the structure or function of the target system, or both. Structure often refers to parts of the target system and how they are connected [3, 28, 70, 74, 84, 86, 93, 96]. When discussing abstract conceptual target systems such as programming concepts, the structure can also refer to concepts related to the target system and the relationships between them [97]. Function refers to knowledge of how and why the target system works [11, 31, 43, 59, 67, 71, 77, 93, 95].

5.2.2. General characteristics

Some studies say that mental models are of varying quality. This refers to descriptions of mental models as representing the reality of the target system with varying accuracy. Mental models are also described as varying in viability [42, 64], completeness [33, 89], and strength [31, 84]. Mental models have also been described as something that can be naïve or based on incorrect assumptions.

Mental models are described as dynamic. This includes notions of changing and updating mental models, as well as the notion of mental models as temporary [28, 57].

Mental models are also said to be affected by the characteristics of the comprehension situation [28, 33] or activated by the specific situation. They have been described as subjective [50] and different between novices and experts [33, 74, 82, 98]. However, some studies equally describe mental models as similar between programmers.

Other attributes of mental models present in the data include runnable [79], hierarchical [33, 76, 98–100], and as describing something abstract.

5.2.3. Structural characteristics

The structure of mental models has been described in two ways. A recurring theme is mental models as a layered structure that provides alternative views of the target system at different levels of abstraction [35–37, 71–73, 75, 94, 99–106]. Another recurring theme is mental models as organized collections of knowledge, describing mental models as containing or consisting of collections, groups, or clusters of related knowledge organized according to some criteria or criterion [30, 33, 55].

Mental models are also described in terms of their relationships with the target system, and some descriptions highlight the mapping between the target system and the mental model or the mappings between the different layers of abstraction present in the model [79, 94, 98].

5.3. Mental model acquisition

5.3.1. Acquisition contexts

Acquisition contexts described in the literature include different learning and sense-making situations. The literature describes educational settings [56, 64, 107], task contexts such as program design [58, 108], and sense-making situations such as learning to use a new system [109].
In program comprehension, programmers’ understanding of programming concepts such as recursion [2,42,43,66] or, more specifically, the structure of the program or the recurring patterns found within the source code [98,110].

Supporting material such as documentation and task specifications was also referenced in the literature [55,61,116].

The programmer’s skills and experiences were also mentioned. Studies stated that mental models are informed by the programmer’s background knowledge, skill level, and experiences [11,17,40,81,108,117–119].

5.3.3. Information sources

Mental models were commonly described as being informed by the combination of information within the environment and the skills and experiences of the programmer [40,80]. They were commonly described as being informed by the source code of a program [55,98,115] or, more specifically, the structure of the program or the recurring patterns found within the source code [98,110].

Supporting material such as documentation and task specifications was also referenced in the literature [55,61,116].

The programmer’s skills and experiences were also mentioned. Studies stated that mental models are informed by the programmer’s background knowledge, skill level, and experiences [11,17,40,81,108,117–119].

6. Synthesis of existing results

To address RQ3, we synthesized the existing results related to different aspects of programmers’ mental models and their acquisition.

6.1. Background knowledge and mental models of programming concepts

Background knowledge refers to a programmer’s existing knowledge about different topics before beginning to comprehend or create a program. It plays an important role in program comprehension and programming.

In program comprehension, programmers’ understanding of programming concepts such as recursion is referred to as a type of background knowledge. However, in computing education, a programmer’s understanding of a programming concept, such as recursion or object-oriented programming, is often referred to as their mental model of the concept [2,42,43,66]. Both terms refer to a programmer’s understanding of some programming concept, and in this section, we discuss results related to both mental models of programming concepts and conceptual knowledge together.

We identified the following types of background knowledge that have been studied:

1. Knowledge of programming concepts refers to knowledge of abstract programming concepts such as recursion [2,42,43,66]. In some studies, this is referred to as the programmer’s mental model of the concept.
2. Knowledge of programming patterns or plans refers to knowledge of reoccurring code sequences that implement some functionality [17,35,36,68,83,94,106,120–125].

3. Knowledge of the programming language syntax refers to knowledge of how these abstract concepts and patterns are implemented in different programming languages [30,33,68,83,84,94,98,109,112,113,116,118,120,126,127].

4. Domain knowledge represents knowledge of common architectures of programs in different domains [128] and programming concepts related to these domains [31,97,129].

5. Knowledge of abstract high-level programming concepts represents programmers’ perception of abstract concepts such as encapsulation [130], coupling [80,131], or object-oriented programming [81,132,133].

Studies suggest that knowledge of recurring programming patterns develops through practice, and is acquired as the programmer develops solutions to different programming problems [123–125]. Acquisition of mental models of programming concepts has mostly been studied in the context of computer science education. In computer science education, students are taught concepts such as variable assignment [43,44], recursion [42,134,135], concurrency [107], and state [136]. Students also learn of higher-level abstract concepts such as object-oriented programming [132,137]. In some situations, conceptual and pattern knowledge is transferred from one domain to another. Conceptual knowledge from, for example, the domain of mathematics can be applied to understanding code, and knowledge from one programming domain can be transferred to another [42,43,97].

In program comprehension, programmers use background knowledge to establish expectations of programs [31,36,37,112,129], and to detect and interpret meaningful code structures by mapping between code and semantic knowledge of the program function [31,68,112,116,118,138].

Background knowledge plays an important role in program generation and building mental models of programming solutions. Conceptual knowledge of concepts such as recursion helps programmers implement solutions using those concepts [82,123,139,140]. Recurring programming patterns are also used in programming. These patterns can be retrieved from memory as required by the task and then used to guide the formation of a solution [50,92,123,125,139,140].

Domain knowledge has been shown to help in programming tasks and especially in program design [92,100]. Knowledge of common concepts, patterns, and architectures within a domain appears to aid in the design of new programs for that domain [69,92,100].

6.2.1. Attributes of mental models of programming concepts

In studies on programmers’ mental models of programming concepts, they have been shown to have certain reoccurring attributes. An attribute that has been highlighted when studying novices is consistency, which refers to the ability to generalize between situations and recognize the same abstract concept in multiple concrete code representations [56,141,142]. We can also think of Mental Models of programming concepts in terms of viability, or how accurately and completely mental models represent the reality of the target systems [42–44,64,66,67,109,135].

6.2. Mental models of programs

6.2.1. Information content and information needs

To understand the information content of programmers’ mental models of programs, we can analyze their information needs, or the types of information they seek about a program. Much of the existing research divides it into two general categories: implementation-level information about the code syntax and control-flow, and higher-level functional and structural information about the program.

At the higher level, information needs include program structure and function [11,143]. We refer to understanding structure and function as understanding the program architecture. This includes the composition and interaction of functional elements that comprise the program [11,144], their role and function [104,105,143], the relationships between them, and their interactions at run-time [11,143,144].

At the implementation level, information needs include the implementation of functionalities [11,78,105,143], i.e., a comprehensive understanding of how a feature or functionality is implemented [11] and behaves during execution [78,94,143,144].

Although programmers have been shown to seek both implementation-level and abstract functional and structural information [35,36,103,104,143,145], the prevalence of either type varies. The level of understanding required is affected by the task at hand, and studies have shown that the level of abstraction at which code is understood depends on the task requirements [143,146]. Knowledge at the implementation level appears to be acquired only as needed and is not always required [11,143].

Some studies have shown that programmers seek information that does not fit these two categories. Programmers have been shown to seek information about what elements are relevant to the task at hand and where they are located in the codebase [78,147]. Understanding the rationale behind the program’s implementation is a recurring theme in the literature as well [11,143,148]. Temporal understanding of a program or understanding past and ongoing development activities and changes in a program appears to be also relevant in some situations [11,143].

6.2.2. Program comprehension

Program comprehension is the process of constructing a mental model of a program. Studies have shown program comprehension as an active process consisting of sequences of actions to gather information about some feature(s) or area(s) of a program according to goals and information needs [36,37,94,104–106,145,147–151]. During the process, programmers use information from multiple information sources both within the code itself and from other sources such as the Internet, documentation, and other programmers to fulfill their information needs [77,143,145,152].

Programmers use both hypothesis-based top-down processes and inference-based bottom-up processes, switching between them based on their current knowledge and their information needs for the current goal [36,37,105,151]. Both top-down and bottom-up processes can contain different actions to locate relevant code elements and extract and comprehend the information present in them. Actions to locate relevant code elements include searching for specific code elements to examine [78,148,150] and scanning the code to find specific code elements or indicators [78,100,103,150].

Actions to extract and comprehend information present in the code include: (i) Reading code line by line [105,145]; (ii) mentally simulating code execution at various levels of abstraction to understand code execution and interactions and relationships between code elements [31,68,100,105,144,147,148]; (iii) reasoning about code based on prior knowledge [32,147] to form hypotheses and explain or interpret code that has been encountered; (iv) running the code to examine it at run-time [77,143,150]; (v) and use of a debugger to examine code execution [78,148].

The comprehension process varies in scope, or how much of the codebase is comprehended [36,77,86,144,152–154], and direction, or if the comprehension process proceeds from a higher-level hypothesis to seeking information or from comprehending smaller code structures to building a higher-level hypothesis [143,154,155]. Some factors that affect the process include the programmer’s goals for the comprehension situation [36,152], the type of code being comprehended, the programmer’s overall level of programming experience [61], and their experience with different types of code [76,91,153], as well as the task at hand [154,156].
6.2.3. Hypotheses in program comprehension

Hypotheses are important drivers of program comprehension. Programmers generate hypotheses about the features, structure, and function of a program they are comprehending [17,35,36,101,103–105,119,129,145,150,152,157].

High-level hypotheses about the function, structure, and functional elements of a program are used to guide further comprehension efforts [103,105,145,150]. Hypotheses about which code areas are relevant for the task at hand, [119,150,152] and where these are located [100,105] guide the programmers’ comprehension efforts towards relevant elements.

Hypotheses can be in the form of expectations about the code that programmers build based on their prior knowledge and then seek to confirm [129,147]. Familiarity with the program domain results in more high-level expectations [116,129,145,158].

Hypotheses can also be inferences based, or assumptions about some feature encountered in the code formed from its indicators [129,147]. These are hypotheses about what the indicated code artifact is or does, and are answered by examining the artifact to confirm or reject the hypothesis [129,147].

6.2.4. Indicators in program comprehension

Indicators play an important role in program comprehension. They are features that indicate the presence of familiar structures [60,111,113,129,138,159,160]. In the literature, these have often been referred to as beacons and cues, although the vocabulary varies between studies.

The main types of indicators in the literature are semantic hints and focal lines. Semantic hints to program functionality are indicators such as meaningful variable-, function-, and file-names, comments, and other elements [60,103,143,152,161–164]. Focal lines are the lines of code most indicative of a familiar programming sequence and can be used to recognize that programming sequence [112,113,120,161].

Indicators have been shown to benefit the comprehension process in two ways. Indicators suggesting that a familiar structure is present in the code can provide the basis for establishing expectations about the code [152,161,165]. Indicators can also be used to confirm hypotheses [116,138,163].

6.2.5. Novice-Expert differences

Novice and expert programmers differ in their ability to comprehend programs. In maintenance tasks, novice programmers have shown to have difficulty understanding the program they work on [166]. Differences in comprehension ability have also been detected between experts and so-called super-experts, or particularly high-performing programmers [167].

Although the results vary between studies [124], most of the research points to the fact that novice programmers struggle to understand the abstract, semantic features of the programs more than experienced programmers do [74,75,99,100,146,147,168]. In general, novices seem to have difficulty to build a higher-level abstract representation of the structure and function of programs [74,75,99,100,169,170].

These differences in the ability to understand program semantics may result from differences in background knowledge. Novice programmers show less knowledge of code patterns, which could hinder their ability to match semantic information with code or problem representations [60,74,77,92,123,147]. Novices’ knowledge appears to be more syntactically based [122,169,171]. Experts, on the other hand, seem to think of concepts they detect in code in more abstract and semantic terms [2,61].

Furthermore, novices can have misconceptions about the semantics of familiar programming concepts [100,128] and the functionality of programming statements [109]. These misconceptions may affect their ability to understand a program [10].

Experts have been shown to detect familiar programming patterns in code better than novices [60,76,98,100,138,147,171–174], and demonstrate a better ability to detect indicators in code [60,147,161]. Experts’

6.2.6. Effect of programming language and programming style

One factor in performing programming tasks is the ability to construct mental representations of the coding solution to be implemented [50,108]. Programmers have been observed creating multi-sensory, dynamic mental representations of programming solutions. The form of these representations seems to be highly individual, and they allow for internal representation of the solution at multiple levels of abstraction [87,90,176]. These mental models represent the programmer’s understanding of the solution or the program to be implemented [92,108] and seem to allow the programmer to mentally simulate the program to allow them to understand how it would function [92,108].

6.3. Task and solution mental models

One factor in performing programming tasks is the ability to construct mental representations of the coding solution to be implemented [50,108]. Programmers have been observed creating multi-sensory, dynamic mental representations of programming solutions. The form of these representations seems to be highly individual, and they allow for internal representation of the solution at multiple levels of abstraction [87,90,176]. These mental models represent the programmer’s understanding of the solution or the program to be implemented [92,108] and seem to allow the programmer to mentally simulate the program to allow them to understand how it would function [92,108].

6.3.1. Information content of solution mental models

One factor in performing programming tasks is the ability to construct mental representations of the coding solution to be implemented [50,108]. Programmers have been observed creating multi-sensory, dynamic mental representations of programming solutions. The form of these representations seems to be highly individual, and they allow for internal representation of the solution at multiple levels of abstraction [87,90,176]. These mental models represent the programmer’s understanding of the solution or the program to be implemented [92,108] and seem to allow the programmer to mentally simulate the program to allow them to understand how it would function [92,108].

6.3.2. Acquisition of solution mental models

When designing a new program, mental models can be created based on information on the requirements and technologies of the new program collected from multiple sources, including customers and team members, and the review of similar systems [8,69]. The creation of these mental models of solutions seems to start from an abstract representation of the key aspects of the solution [8]. This is then systematically expanded to a more concrete level [70,92,123,178,179]. Some studies hypothesize that the initial kernel idea for the solution comes from the background knowledge of similar programs [123,179], which is then systematically expanded to suit the current problem situation.
6.3.3. Task context models

In programming situations where the programmer works with an existing codebase, they create mental representations not only of the current programming task but also of the program on which they are working [3,40,92,157,167]. In some studies, this also includes a mental representation of the problem or an understanding of the problem for which the solution is implemented [50]. In program design situations, representations of the problem domain have also been observed, but here in the form of understanding the abstract problem domain where the program will operate [69,157]. The methods for comprehending programs in different task situations do not seem to differ from general program comprehension methods, which have been discussed in previous sections.

6.3.4. Novice-Expert differences

Novice-expert differences in the ability to build abstract mental models can be observed in programming tasks such as program design, where novice programmers struggle to construct abstract mental representations of the programs they are designing [180]. As programming experience increases, so does the ability to represent tasks, programs, and problems as larger abstract units [50]. Beginner programmers have been shown to meticulously translate program statements into code step by step, whereas more experienced programmers show the ability to represent elements of both problems and solutions as larger abstract units [50,85,92,123]. When comparing programmers of different experience levels in modification tasks, one of the attributes of an expert programmer is their ability to create an abstract and complete representation of the program to be modified [61,181] and the task at hand [85]. Experienced programmers also have a better ability to use recurring programming patterns in programming [83,100,182].

7. Discussion

In this section, we discuss the answers to our research questions.

7.1. How have programmers’ mental models been studied?

In this review, we analyze the state of the research field on programmers’ mental models. We evaluate the coverage of research in terms of target systems and participant populations. Additionally, we evaluate the study designs and the theoretical concepts used in the research.

7.1.1. Participants, experience, and target systems

Our findings suggest the need to assess how modern software development settings affect programmers’ mental models. Many of the studies that have provided the basis for the current research on programmers’ mental models were conducted over a decade ago, and thus used programming languages such as BASIC and Fortran. Over the years, programming languages and tools have developed, and new programming languages have risen to prominence. Modern programming also relies heavily on different APIs and complex programming environments. Naturally, older studies do not account for all of these technologies.

In terms of study participants and their experience level, our results show a good balance between experienced and novice participants being used. However, we categorized study participants according to the statements in the studies. Therefore, we cannot guarantee the comparability between the classifications.

The most common type of participant was student. Some studies in the result set aimed to understand students’ mental models, so the use of student participants is expected. However, in some studies, students were used instead of or representing programmers in general. This practice, while being supported by parts of the research field, has also been debated [183,184].

7.1.2. Study designs

Our results highlight the need to validate existing research results in contemporary real-world settings. Most of the included studies were carried out in laboratory settings. Research in laboratory settings is critical for understanding a phenomenon [185]. However, research in natural settings is required to evaluate the applicability of the results in real-world situations.

Non-coding tasks, or tasks where participants completed paper or computer-based tests, were the most prevalent study tasks. These studies have been critical in understanding programmers’ mental models and their acquisition. However, to understand programmers’ mental models in real-world settings, additional research using realistic tasks is important. Researchers have made similar arguments about the field of software engineering research in general, calling for realistic study designs to confirm the applicability of research results in real-world settings [186].

7.1.3. Theoretical concepts

In general, the use of theory in software engineering research is lacking [187,188]. This lack of background theory can cause problems in interpreting the study results, as no framework is used to explain and conceptualize the results [15]. Our results show that this is the case in research into programmers’ mental models as well. While a large body of literature exists on programmers’ mental models, a large proportion of the studies did not rely on any theoretical concepts (see Fig. 6).

The studies that relied on different theoretical concepts showed the scattered nature of the research field. The structure concepts and acquisition concepts that we identified from the initial set of studies were not enough to cover the wide set of concepts and theories present in the result set. The “other” category was large for both theoretical concept categories, even after adjustments were made during the snowballing phase. When analyzing the field of software engineering research, similar results have been reported, showing little shared theory between studies [187].

7.2. How have programmers’ mental models been conceptualized and described?

With RQ2, “How have programmers’ mental models been conceptualized and described?” we sought to analyze how mental models are defined in existing research. Although our results show that the general understanding of programmers’ mental models is somewhat unified across the research field, the specifics of programmers’ mental models and their use and acquisition remain undefined.

There is variability in how mental models are defined. While we can conclude that programmers’ mental models are considered to be meaningfully organized representations of programmers’ understanding of aspects of a target system, the definitions show a divide between mental models as representations of understanding or learned knowledge, and mental models as cognitive structures used for reasoning and prediction. This variability is also apparent in descriptions of the areas of memory related to mental models, which varied from working memory or short-term memory to papers taking no stance on the memory location. This variability in memory locations was already discussed by Canas in 2001 [20].

Although there is variability in how mental models are defined, their uses are understood in similar ways across the research field. According to the research field, mental models are used in performing tasks related to the target system: they allow explaining, predicting, and interpreting the target system, thus aiding in tasks such as making decisions and predictions. Mental models are described as mental representations that are used when performing programming tasks such as debugging and modification. They are described as means to understand, interpret, and explain the target system through mental simulation and reasoning and that programmers use to make decisions and predictions. This
understanding of mental models reflects the understanding of mental models in cognitive science [4].

The descriptions of how mental models are acquired vary between studies, reflecting the different definitions of mental models. Our results show that mental model acquisition is understood to be not limited to program comprehension, but they are acquired in various sense-making situations, such as educational settings, program comprehension, program design, and learning to use a new system. This process is described as iterative and active, using the programmer’s background knowledge. This acquisition process and the resulting mental models were described as being informed by a combination of information present in the environment and the programmers’ knowledge and experiences. In particular, the program code and other auxiliary materials, such as documentation, were mentioned as information sources. We can synthesize the descriptions as mental model acquisition is understood to happen in various situations where the programmer makes sense of the target system. This process is iterative using the information available in the situation and the background knowledge and experience of the programmers.

7.3. Synthesizing existing knowledge on programmers’ mental models

In terms of program comprehension and mental models of programs, our results mostly correspond with Von Mayrhauser’s integrated metamodel [38] and the schema-based model [7,31,68]. Von Mayrhauser’s theory detailed program comprehension as an active, goal-driven, hypothesis-based process that utilizes both top-down and bottom-up methods, switching between them as required [17,35,37,38,104]. As discussed in previous sections, her theory and results related to it are also supported by other studies in our result set. Our synthesis points toward program comprehension being a combination of top-down and bottom-up processes driven by hypotheses, goals, and the task at hand, aiming to gather semantic and syntactic information about the program simultaneously.

Our synthesis also shows partial support for schema-based top-down program comprehension, which details top-down methods of program comprehension and the use and creation of hypotheses [7,31]. We hypothesize that top-down program comprehension, as observed by Von Mayrhauser, could be explained by schema activation processes and the use of schemata as posited by studies on schema-based program comprehension [31].

According to our synthesis, programmers’ information needs and information contained in mental models may contain types of information that were not included in previous theories of programmers’ mental models. Based on our synthesis, programmers’ mental models are mental representations of knowledge about the target system. When it comes to mental models of programs, we see the need for both syntactic and semantic information about the program, although the prevalence of either type of information depends on the programmer, their level of expertise, and the specific sense-making situation. Although this result is in line with existing theories of mental models [7,21], existing theories limit the semantic elements of a program to understanding the goals or functions of the program and its links to the problem domain [21]. Our synthesis reveals that high-level program understanding may include other elements as well. In studies discussing situations in which a programmer works with a program for an extended time, studies described programmers’ mental models as containing a view of how a particular system develops and changes over time [11,143]. Other studies discussed the rationale of the program or as an important information need, even naming it as a layer in the mental model itself [11,143,146].

Many studies also provided results that suggest that there may be a spatial aspect to programmers’ mental models. In our analysis, we see that the location of specific code elements is an important information need. Programmers search and scan code to find specific code elements. They also create hypotheses about their location based on indicators and their background knowledge [78,100,147,148]. However, we must acknowledge that our results cannot provide any insight into whether this spatial information about code locations is stored as a part of the mental model, or as some separate schema or other knowledge structure.

Different theories describe the structure of programmers’ mental models in different ways. According to the integrated metamodel, the end result of program comprehension is a two-layered mental model that represents the semantic and syntactic aspects of the program as different layers of the model [17,35,37,38,104]. Schema-based theories describe the end result of comprehension as a set, network, or hierarchy of activated schemata [7,31]. Our synthesis cannot provide insight into the exact nature of mental models to refute or confirm either view of mental models.

We conclude that while the existing theories on programmers’ mental models capture many important aspects of programmers’ mental models, these theories need to be reevaluated when it comes to the importance of locating code and the information needs of programmers in real-world programming contexts.

7.4. Threats to validity

The design of the search string could lead to the omission of relevant articles. The search needed to catch relevant articles, and simultaneously omit irrelevant results. We opted for a broad search string and relied on our exclusion criteria to remove irrelevant results. We also complemented the database searches with backward and forward snowballing, as discussed in Sections 3.1.2 and 3.4.

Our article selection and filtering process is a source of potential bias. During database searches, a researcher conducted the first exclusion round. Care was taken to exclude only obvious false positives, such as results not written in English, and results that were not the required publication type, such as posters or abstracts. In the other phases of the process, we followed well-established guidelines [189]. As discussed in Section 3.2, the filtering was done by a team of researchers and a verification step was used to mitigate the biases of any single researcher.

The data extraction process is another source of potential bias. To ensure consistency in data extraction, we first ran a round of practice extraction sessions, as discussed in Section 3.5. Furthermore, we used a verification process described in Section 3.5 to mitigate the biases of any single researcher. Our emphasis was on detecting and repairing inconsistencies in extraction, not measuring extraction errors. Therefore, we did not calculate the interrater agreement between extractors.

Our data extraction includes data that was summarized from the articles by each extractor, such as the mental model description data. These summaries can be subjective, biasing the results. We used thematic analysis [54] to analyze descriptions of mental models and summaries of results related to mental models. Two researchers participated in the analysis to mitigate some risks posed by the subjectivity of the researchers. The results of the analysis were presented to the whole research team for validation. Given the extraction verification process and the way the data was analyzed, we argue that the reliability of the data is adequate for our purposes.

Finally, due to the large number of papers included in this literature review, this study cannot cover all the individual results in much detail. Instead, we aimed at synthesizing existing work to help organize the disconnected areas in the field of mental model research.

8. Conclusions

This systematic literature review assessed the research on programmers’ mental models to bring consistency into the fragmented knowledge base related to them. The importance of researching programmers’ mental models to understand and support their work has been recognized by the research community. Different tools and techniques have been created to aid programmers in developing accurate
mental models [9,12,13]. However, the efficacy of these tools and their adoption into wider use has been lacking [9,13,14]. One reason for this may be lacking understanding of programmers' mental models, and there have been calls for further research into programmers’ mental models and their development [1,13]. This review answers those calls by providing a comprehensive synthesis of existing results related to programmers' mental models.

Programmers’ mental models have so far been studied from mutually inconsistent starting points, producing a largely scattered set of findings. This fragmentation in the findings and the lack of a coherent theoretical basis cause difficulties in advancing the research field into analyzing and interpreting comprehension and generation of large-scale software in modern settings.

The majority of the existing research has used non-programming tasks, where code cannot be run or worked with as usual, in laboratory settings. Furthermore, these studies have not considered many modern programming languages and tools. These studies have provided important information on programmers' mental models and their acquisition. However, our results indicate that more research is required to explore how the existing results apply to contemporary settings to achieve an understanding of programmers' mental models that applies to contemporary real-world settings.

In this work, we have untangled some of the fragmentation in the field. Our analysis shows that there is consistency in how mental models are conceptualized and described across the research field. They are conceptualized as meaningfully organized representations of programmers' knowledge of some aspects of a target system and are used in performing tasks related to the target system. We also provide a synthesis of results related to programmers’ mental models. We argue that this synthesis can be used to build a shared understanding of mental models to use as a basis for further research and to provide a basis for developing tools and techniques to support programmers in their work.

CRediT authorship contribution statement

Ava Heinonen: Conceptualization, Methodology, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization, Software. Bettina Lehtelä: Methodology, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing. Arto Hellas: Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. Fabian Fagerholm: Methodology, Validation, Investigation, Data curation, Writing – original draft, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We thank Balázs Nagyvárdai, Jehan Khattak, and Markus Laattala for their help in initial study filtering and data extraction.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.infsof.2023.107300.

References
