Understanding APIs and the software that provides them - Analysis of programmers' API mental models used in programming tasks

Ava Heinonen
Department of Computer Science
Aalto University
ava.heinonen@aalto.fi

Abstract

At one point in time, programming could be thought of as the act of translating program requirements into source code written in one programming language. However, modern programming relies heavily on building programs by integrating services, functionality, and data provided by external software into a program using APIs. In part, programming consists of selecting suitable software that provides ready solutions for programming problems and using their APIs to integrate those solutions into a coherent program.

This change in programming necessarily changes programmers' mental models — their understanding of the programs they work on. In this paper, we discuss programmers' mental models when using an API in a programming task. We conducted interviews with twelve industry professionals using the critical decision method. We analyzed the mental models — the understanding these practitioners utilized when completing different software development tasks using an API. Through this analysis, we were able to identify information about the tasks, the software providing an API, and the API that were represented in the programmers' mental models. These results contribute to the existing literature by opening a discussion on how using APIs changes the nature of programs and programming and by providing insight into the understanding necessary for completing programming tasks using APIs.

1. Introduction

Modern software development can be understood as a process of integration, combining software components into an overall system. Software libraries and frameworks provide pre-implemented solutions for many programming problems, which can be integrated into a new system by calling methods in their APIs. Services hosted on the Web can also be utilized as part of a software system by making requests to their API endpoints (Mäkitalo, Taivalsaari, Kiviluoto, Mikkonen, & Capilla, 2020).

This shift in software development has changed the nature of computer programs. In the past, a program could be thought of as a specification written in a programming language that expresses a set of calculations (Pair, 1990; Détienne & Détienne, 2002b). A program was a set of calculations and objects, expressed in a programming language with rules on how to organize words into meaningful expressions (Pair, 1990; Détienne & Détienne, 2002b). While a program is still a set of calculations and objects, the way these are expressed is different. A program now comprises objects and calculations expressed as code written in a programming language or implemented elsewhere and accessed via API calls. The meaning of these calls is defined not by the programming language itself but by the programs implementing them. A program is a set of calculations and objects, some implemented in the program's source code and some by other programs. These are expressed either as source code or as API calls that indicate the execution of externally implemented calculations.

This shift in what a program is necessarily changes the way programmers design and implement programs and, even more so, the knowledge and understanding required to do so (Andrews, Ghosh, & Choi, 2002). As a programmer designs and implements a program, they form an internal representation of it — a mental model that represents the program to be implemented (Heinonen, Lehtelä, Hellas, & Fagerholm, 2023; Kim, Lerch, & Simon, 1995). This understanding of the program to be implemented then guides the process of translating the program design into an executable program (Pennington & Grabowski, 1990). However, the discussion around programmers' mental models has focused on programs as speci-

fications written in a programming language that expresses the calculations of the program (Détienne & Détienne, 2002b; Pennington & Grabowski, 1990; Heinonen et al., 2023).

Programming also requires the programmer to translate the designed solution into an executable program (Pennington & Grabowski, 1990). The shift in software development has transformed this task from writing code that implements a functionality to writing code that interacts with the interface of a program that implements a functionality (Andrews et al., 2002). This process can be described as writing "glue" code or writing client code that integrates functionality from different software components into one program (Chen, He, Liu, & Zhan, 2007).

In some ways, writing client code is similar to software reuse, where the programmer must understand what type of solution is required, find a suitable solution, and adapt it to fit the target software (Détienne & Détienne, 2002a). However, reuse theories do not account for the differences between reusing a piece of code and integrating a program into another using its API. The latter requires not only understanding the solution and its suitability, but also another level of understanding: the interface through which the solution can be accessed (Thayer, Chasins, & Ko, 2021; Mosqueira-Rey, Alonso-Ríos, Moret-Bonillo, Fernández-Varela, & Álvarez-Estévez, 2018). Successfully using an API also requires the programmer to form a mental model of the software providing the API — what it is, what it can be used for, and how it can be used in a program (Heinonen & Fagerholm, 2023).

As the nature of how programs are expressed, developed, and understood is changing, theories and research on the cognitive aspects of programming should also evolve. Theories of programmers' mental models should encompass the concept of a program as an integrated system of multiple programs and consider how programmers conceptualize the external programs integrated into their own programs. Furthermore, programming theories need to address the cognitive aspects of using APIs. In this work, we will contribute to this endeavor by presenting results on programmers' mental models of APIs and the programs that provide them. We will also present findings on how these mental models are utilized in programming tasks.

In everyday discussions, different libraries, frameworks, packages, web services, and other pieces of software that provide an API are all referred to as APIs. However, this terminology does not allow for differentiation between the software that provides an API and the API itself. In this work, we will refer to software libraries, frameworks, services, and other software that provide functionality, data, or other resources that can be integrated and utilized in a new software system as *provider software*. We will refer to the functionality, data, and services that the provider software offers, which can be utilized in building new software, as *resources*. Finally, we will refer to the interface provided by the provider software that allows a program to utilize its resources as a *API*.

In this work, we aim to provide insight into programmers' mental models as they design and develop programs that utilize a provider software's resources using its API. We present results from a study where 12 professional programmers were interviewed using the critical incident method about a situation or situations where they had to learn to work with a new provider software. We analyze the mental models of the participants through their expressions of their understanding, through the problems they expressed having with understanding some aspect of the task and its context, and through the information they sought and used to complete the tasks. Through this analysis, we identified some key aspects of programmers' mental models of provider software. We also identified some key mechanisms of how these mental models are developed and how they are used in completing programming tasks.

2. Methodology

To conduct the study, we used the critical incident method interview protocol, designed to elicit information about cognitive performance in complex task settings (Marcella, Rowlands, & Baxter, 2013), which has been used successfully in similar studies (Votipka, Rabin, Micinski, Foster, & Mazurek, 2020).

We conducted 12 interviews in which we explored programmers' mental models of provider systems and their resources and APIs through detailed analyses of one or more recalled real-life events where

the participants had to learn about a new provider system.

2.1. Participants

A total of twelve participants took part in the study. Our participants were employed in different roles in academic and industry settings. Two participants held primarily academic positions, while ten were employed in software development companies or software development teams within academic or other organizations.

2.2. Interview protocol

During the interviews, participants were asked to choose an event in which they had to learn a new API and recall the process while we drew a diagram to visually represent it. Throughout the interviews, the interviewers asked further questions about multiple items of interest.

The interviews lasted approximately 60 minutes and were conducted remotely or in person, depending on the availability of the interviewee. The interviews were divided into three parts: background questions, event selection, and event walkthrough, as described subsequently.

2.2.1. Background questions

At the beginning of each interview, participants were asked about their education, programming background, and the level of experience they have with the technologies they currently work with. They were also asked about their current job and the tasks and responsibilities of their current role.

2.2.2. Event selection

After the background questions, the participants were asked if they could recall a time when they had to learn to use a new API. They were prompted to think about a memorable event, either recent or otherwise memorable to them. If a participant had difficulty recalling a suitable event, we asked further questions to assist in event selection.

2.2.3. Event walkthrough

After a suitable event had been selected, the participants were asked to recount what had happened during the event. As participants described the event, the lead interviewer drew a diagram of the process described by the participant. The participants could see the diagram at all times and were asked to notify us if the diagram did not match their story in some way.

While participants described the event, we asked directed questions intended to clarify some aspect of the event or gather further information about some important or interesting aspect of the event.

2.3. Data Analysis

We utilized iterative coding to analyze the data with the goal of examining programmers' mental models as they engaged in programming-related activities that required them to learn a new API.

In the first round of analysis, we coded items related to the programming projects the participants were undertaking and divided the interviews into activities, such as designing or implementing a solution.

In the second iteration, we analyzed each activity in detail. We coded statements about mental models, knowledge gaps, sensemaking activities, and resulting understanding. In this article, we will present results related to mental models. Our analysis of knowledge gaps and sensemaking activities will be presented in another publication. Statements about understanding or perception of some aspect of the task or its context were coded as mental models.

During the third iteration, we analyzed statements related to mental models to identify the aspects of the provider software, its resources and APIs, and the programming tasks that were represented in the mental models.

3. Results

In this section, we will discuss our results related to programmers' mental models of provider software and their resources and APIs. We will first examine programmers' mental models of provider software, followed by the mental models of programming tasks, API resources, and APIs used when implementing

a program that utilizes a provider software through its API.

3.1. Understanding provider software

A programmer's mental model of the provider software represents their understanding understanding of the software — what it *is*, what it can be used for, and some relevant quality attributes such as usability and quality. We have divided the results into two categories. The first category, "What it is and what it can do," represents an understanding of the type of the provider software, its function, and functionality. "Quality, usability, and other relevant attributes" represent different non-functional characteristics of the provider software.

3.1.1. What it is and what it can do

These aspects of the provider software represent the programmer's understanding of what a provider software is and what it can be used for.

Type refers to the kind of provider software a specific software is — whether it's a library, a web service, or something else. Knowledge of the type of a provider software allows the programmer to utilize their background knowledge of other provider software of the same type to understand what the provider software is guiding the process of learning about and using a provider software.

As our participants were professional programmers, they had previous experience with different types of provider software. We refer to this knowledge as *provider software type schemata*, which represent different types of software, the functionality those types have, and how they can be used. For example, a schema of REST APIs suggests that they provide access to data in a database and can be used by sending HTTP requests to API endpoints.

Our participants discussed provider software as instances of types. The recognition of a software's type provided them with expectations of how it can be used, such as expecting that a REST API is used by sending HTTP requests or a library is first installed to the project and then used by calling API methods. It also guided the participants forward, providing expectations of what to do next and what information was needed. For example, one of our participants discussed finding a suitable provider software, and knowing that a library has to be installed, moved on to seeking information about how the specific provider software could be installed using Cradle.

Function refers to the general purpose of the provider software. This was discussed as the type of task that could be done with it, such as "draw plots" or "access data from a database." For example, one of our participants described PNPM as follows:

> It is used to install react native...or of course you can use it to install anything. So a PNPM package manager built on top of NPM. And it can be used to install all kinds of JavaScript dependencies.

This understanding not only aids in selecting suitable provider software for the task at hand, but it also provides the programmer with expectations about it. There are often many provider software with the same function, and they have some similarity between them. We refer to the knowledge of these similar provider systems as *provider software category schemata*, which represent knowledge of provider software with the same function, such as what libraries that provide methods for making HTTP requests generally are like. Based on our interviews, participants used their provider software category schemata when encountering a new provider software with the same function. These schemata provided them with expectations about the software in question — its use, functionality, and even the naming of API methods:

> ... I could already guess the name of the method I could use. Because with these the naming of the methods is quite similar. They are usually always named the same way. For example something like findAll, findOne, findByID. So the query abstractions are usually always named like that...

Functionality is the set of resources provided by the provider software. These resources include data, services, and implementations of behavior. Programmers' understanding of provider software functionality enables them to select a provider software suitable for the task at hand, as described by one of our participants who needed a provider software that provides tab components:

> At that point I had read the MaterialUI documentation quite a bit. It has all kinds of examples, and I have scrolled the sidebar which lists all the components that it has, and I had considered using it [the tab component] previously but I had not previously needed tabs for anything.

However, programmers' understanding of a provider software's functionality does not always appear to be comprehensive or entirely accurate. For instance, one of our participants expected that a provider system for drawing plots would provide a way to combine the titles of subplots into one and were disappointed when they learned that it did not:

> I was a bit annoyed that there was no automatic way to do it. You'd think that combining identical sub-plot titles would be a relatively common use case. So then...I expected someone would have made something automatic for it especially since it has so many other automatic features.

3.1.2. Quality, Usability and other relevant attributes

There are, of course, multitudes of attributes a software has, ranging from fault tolerance to availability. However, not all of these are relevant at all times. Quality and usability of a provider software were mentioned as important by multiple participants and are thus discussed in more detail below. Participants also mentioned other attributes when those were relevant for the specific task or project they were working on, indicating that participants considered the attributes that were relevant for them at that time. Therefore, we will not discuss all possible attributes but focus on the notion that the mental model seems to contain some information about the attributes deemed important for the task or project at hand.

Quality refers to the programmer's perception of the "goodness" or quality of the provider software. Our participants talked about forming a perception of whether the provider software was good or "valid" to use. For example, one participant discussed building a perception of a provider software's validity before selecting it for use:

> They [libraries] are almost always open source, so I usually also check its validity as well. So I usually check the GitHub repository to see, for example, if it has been actively updated and if it has a lot of these...umm...these like stars which are kind of like "likes" and if it has a lot of forks and all those kinds of things. Like if it seems like it is used a lot and is like validated by the developer community so it is valid. And that can also peak my interest [in using a provider software] as well.

Usability refers to the programmer's perception of how easy the provider software is to use. When selecting suitable provider software, our participants discussed forming perceptions of the usability of the API. For example, one of our participants described a provider software they liked as "logical, easy to use, and easy to understand," while another participant discussed liking a provider software because it seemed easy to use and understand.

Other relevant attributes: Our participants did not seem to form an understanding of all possible attributes of a provider software, but rather only the attributes relevant to the task at hand. For example, one participant had to consider the performance of a provider software to design a component using it. However, this participant stated that they would not want to need to know this information, indicating they would not have gathered it if it was not necessary:

> Well I would like to think about it logically, in other words so that I would just need to know how to use it. In this case I also had a bit of understanding of its technical side. We had to consider it, mostly its performance, and if it would cause any problems.

3.2. Use of a provider software for a specific task

In this section we will discuss the mental models related using a provider software to achieve a specific outcome. The outcome, of course, is most commonly a program that has a certain functionality.

The understanding related to using a provider software contains multiple aspects of the situation, including what is to be achieved, what is required from the provider software to achieve it, how the required resources are modeled in the provider software, and how those resources can be accessed. Below, we will illustrate this using an example from the interviews, and then we will describe the aspects of the mental models in more detail.

3.2.1. Example

One of our participants was implementing the backend for a mobile application:

> So I had a specific application in mind already. It was a backend for an application that I made for a course. So for the students to use...so in the course the students make a client for the backend...so they have the backend ready but they have to make the mobile application client for it... So it was like an application for users to review GitHub repositories. So the user can log in and then write reviews for a GitHub repository.

They started to implement the backend working on one endpoint at a time. They start working on an endpoint that lists all the repositories. They know that the provider software they are using provides functionality for making requests to a database, and after forming an idea of what kind of a functionality they'd need for the endpoint, they start looking for a suitable method:

> So I knew that I have...I knew that I have a database table where the repositories are, and it has certain fields. So then I started to think that if my endpoint has to list all of them, I started to think about what kind of a method I had to look for.

They browse the API documentation to find a method that seems promising, and read the method description to verify that the method indeed does what they need. They then use a code example in the documentation to write the client code that calls the method:

> It [API documentation] has like API reference, so like the API in a more technical level. So I went there. And that was organized by the main themes, so there was like queries, so a section about how to make queries. And the methods were listed there. And from there I could spot a method that could be the right one. And then based on the method description I verified that it really was. And it also had code, they usually also have code examples showing how to actually do it.

3.2.2. Goals, tasks, solutions and resources

One of the aspects of the programmer's mental model is an understanding of what is to be implemented, and what is required from the API to implement it. This includes the program that is to be implemented, the part of the program the programmer is currently working on, how the part of the program can be implemented, and what is required from the provider software to implement the part.

*Goal software:*refers to the programmer's understanding of the program they are implementing. For example, in the previous example one participant described the backend application they were developing. Our participants described having different levels of understanding of the design of the goal software.

In some cases, they were working from formal design documents that provided them with a detailed understanding of the architecture of the goal software. In other cases, they did not describe the use of formal design documents or processes, but they did have a rather extensive understanding of the design.

> So. We decided that we should make a new component which handles this part of it. So we made a new container for it. And in that, in regular intervals it fires up, and the idea is that it makes a request to the API and fetches... Or actually first it fetches a configuration file from DynamoDB which tells it what to fetch. It specifies what to fetch from where...And then based on the configurations it makes queries to the API and fetches all the information about the campaigns. And and so we know that first it fetches all the campaigns, and checks their timestamps to see if some of them have changed. And if they have, then it fetches information about those campaigns like product information- and then it updates the information to another DynamoDB table.

However, when participants were adding functionality to existing programs by integrating a service that provides the entire functionality, their understanding of the resulting integrated program was limited. This understanding was primarily shaped by their background knowledge of applications in the domain rather than their understanding of the specific provider software. For example, one participant had already began integrating a service into an existing application when they encountered a problem that necessitated them to develop a deeper understanding of the system they were constructing. Their surprise at how the system works indicates they did not possess a robust mental model of it previously:

> So. I hadn't previously like. So rarely some thing from your own code ever calls anything else. So all of our services work so, that a frontend always has one backend. And the frontend always talks to only its backend and then the backend may call some other service or do anything else. And in this case it was like "wait a second, it talks with something else". Our frontend sends requests to it kind of like google analytics...So like after I got it it was more clear that "hey this is what we are doing and this is what it is all about". endquote
>
> *Task:* refers to the programmer's understanding of the specific part of the goal software they are currently working on. When the program is small, the entire program could be the task. However, when the program is larger, it is split into parts, and the programmer works on them one at a time.
>
> In many of our interviews, it was not clear how the participants identified the task at hand. However, participants who discussed tasks related to setting up the provider software could often provide more detail on how the tasks were identified. In some cases, participants identified the task based on their experience with the type of provider software. For example, one participant was using a library. Drawing from their familiarity with other libraries, they recognized they had to add the library to their project and moved on to read the library documentation to learn how to do so. In cases where the participant was integrating a service to an existing program or when a library required more extensive setup, participants also mentioned reading documentation to learn what had to be done.
>
> *Solution:* refers to the programmer's mental model of the program that fulfills the task. Some of our participants described the solution in functional terms - what the program should do.
>
> Other participants described the solution in terms of programming concepts or patterns that could be used to achieve the required functionality. They discussed knowing, for example, that to create a program that fetches specific data from a database they had to make a HTTP GET request, or that to create a program that makes multiple HTTP requests in parallel they should create a new threads. For example one participant was implementing a program, that allowed users to authenticate to a web-service. The participant was working on keeping the

users logged in, and based on their domain knowledge and knowledge of the service, knew that they should use refresh tokens to implement the functionality:

> The situation was, that the authentication worked using OAuth, which means it was token-based. And with tokens it is essential that, for example, you stay logged in. So you don't have to log in again after like an hour when the token expires. And for that you need a system called refresh token.

Some participants discussed how they identified the concept(s) that were required to implement the solution. For example one of our participants was implementing a modification to an existing web application. They knew that the program should allow the user to switch between two views, and it should be easy for the user to see that there are two views available. They then spend some time figuring out the solution:

> I did not know exactly how I wanted to do it. With React it is really easy to switch between components inplace. You just use a conditional statement to remove one and then change the state to show another one. And then switch back. Like a toggle. But I thought that way it would not be so easy for the users to see that there are multiple views there. There at the same place. So then I thought that the other view actually already has tabs for different files, so we could make another level of tabs.

In cases where the participant was already familiar with the provider software, they discussed the solution in terms of the provider software.

Resources: refer to programmer's understanding of what is required from a provider software to implement the solution. Some participants discussed the resources in terms of programming concepts that they expected a provider software would provide the tools to implement. One example of this is the participant who wanted to implement tabs described above. Other participants discussed needing the implementation of a specific behavior, for example fetching all rows of a database table. Participants who were writing a program that interacts with a service, the resources were described in terms of provider software functionality that the programmer wanted to use, such as authentication to the service.

Some of our participants were using provider software to access data, and knew the data that they needed for their programs, such as one participant who was making a program that checks the age of an user account, and knew they needed a datamodel that contained the age of a specific user account:

> Basically we knew the datamodel...We were trying to do when we upgrade the subscription. So basically what happens, we usually the company will draw out the accounts, and sometime the person who is upgrading to the new service is using ten years old account. And then we need to upgrade it to the new account because that Is too old... This is the thing which was, we need to calculate specific years...

The division between solutions and resources is not always clear. Some participants first designed the solution and identified resources they needed from a provider software, and then moved on to figure out which provider software provides the required resources and how to use it to add those resources to their program. Some participants had already selected a provider software to use, and as they designed their solution they did so based on their understanding of the provider software's functionality, so the solution was designed based on how it could be done using the specific provider software they had in mind.

3.2.3. API-translated solution model and its implementation

Another aspect of the mental model is an understanding how to use the API of the provider software to utilize specific resources. This understanding can be roughly divided into two categories: Understanding how the resources are modeled in the API, and understanding how to implement the code that interacts with the API to access the resources.

API-translated solution model: represents the programmer's understanding of how the resources they want to use are modeled in the API. In other words, the API-translated solution model describes the solution in terms of API elements, API tasks, or when it comes to data resources, datamodels. Some participants described the API-translated solution model in terms of *API tasks*. With API task we refer to the set of tasks the client code has to perform in regards of the API. For example, one of our participants was using a provider software that implements a functionality that draws plots. The task they were working on was writing the code that draws a title to the center of the plot. To do so, they first had to figure out how this behavior is modeled in the API. Using StackOverflow they learned that the client code has to first instruct the provider software to remove the titles from all the sub-plots, and then add text to specific coordinates within the plot:

> So you remove from the Seaborn...make it so that it does not make the sub-titles or titles for the sub-plots. And then you just manually write text to it [the plot] through matplotlib using those like coordinates. So you just manually add text there and give it its orientation and coordinates.

Some participants described the API-translated solution model in terms of API elements and their relationships:

> The tab API consisted of a tabs component, and you place tab components within it and give each tab an index. And tabs receives a state I think. And then there is also a tab panel component which is placed within a tab. And then a tab panel is shown according to the state.

When the resources are data, the provider software models information about different entities as datamodels, and the API-translated solution model refers to the datamodels and their attributes and relationships that represent information about a specific entity. For example one participant was working on a program that writes specific data to a provider software. They knew the entity they wanted to write information about. However, to write the program they had to first figure out how the entity was represented in the provider software:

> So our department started to write the information about [entity] into the [provider software] using our own system...So we just checked from the API catalogue that here is this datamodel called [entity] and they can be created using this endpoint here...And then we checked what we needed for the datamodel. So it needs references to three different things. And we wondered how we were supposed to get the references from? So we do have the three other datamodels, but how are we supposed to get the right instances?...At that point it was unclear why in the world of [provider software] the information about [entity] is split into three datamodels and what the relationships between the models are...So we did not know what the three references actually mean. And then when we were told that you have to use this [another datamodel] then it started to make sense like that one had all the information we need so we can start by saying take this one first. At that point the whole thing did not seem so difficult and confusing anymore.

Implementation model: refers to the implementation details of the client code for API tasks, elements, and datamodels. This includes the syntax of the client code as well as their

parameters. This may also include the syntax of configurations or other code that has to be written to implement the solution. For example, one participant described seeking for information about the data type of parameters:

> I checked the tab panel and the tab, the individual tabs, what I have to give them, to make sure I give them the right index, or like what kind of index they need. That was not explained in the code example page.

Participants often described searching for and reading code examples to figure out how to use provider software. In these cases the information about the API-translated solution model and the implementation model were acquired simultaneously. In other cases, participants described learning the API-translated solution model first, and then moving on to seek information about the correct syntax to implement it.

4. Discussion

In this paper, we present the results of a study where we used the critical incident interview technique to gain insight into programmers' mental models of provider software, their resources and their APIs. Our results show, that as programmers work with provider software, they form a mental model of the provider software, that represents their understanding of the provider software as a software artifact — its function and functionality, the way it works and is used in a program, and its quality, usability, and other attributes relevant to the situation.

This mental model provides the necessary framework for utilizing the provider software's resources in a program. As programmers integrate provider software's resources into a program through its API, their mental models of the provider software's type, function, functionality and use guide the process of identifying resources required to solve a programming problem, and figuring out how the resources are modeled in the software and how to integrate the them into a program. This result corresponds with the idea of initial API mental models, that represent programmer's understanding of a provider system they form before they begin using the provider software in programming tasks (Heinonen & Fagerholm, 2023). These mental models guide the programmer's actions, providing them with understanding of what the provider software can be used for and how the provider software can be used, as well as what information is required to use it and what should be done to begin using the provider software (Heinonen & Fagerholm, 2023).

When it comes to program design, our results show that the process is in most cases quite similar to the models of program design that have been previously proposed. Existing theories of program design describe program design as a process of decomposing a problem into manageable sub-problems that can then be broken down further and solved (Atwood, Jeffries, Turner, Polson, & CO, 1980; Pennington & Grabowski, 1990). We see the same behavior when designing solutions that use provider software. Our results show programmers breaking the goal program down into manageable tasks, that can then be solved one by one. However, in cases where programmers integrate services into existing programs or use libraries that require extensive setups, this decomposition of the problem is different. In these cases the programmer does not necessarily have a deep understanding of the resulting program, and thus use documentation to gather the information about the tasks required to implement it.

When writing code, traditionally, we would expect the programmer to understand what is to be implemented, how the required functionality can be achieved in computing terms (Pennington & Grabowski, 1990). We would also expect the programmer to have an understanding of the programming language used to write the code to implement the solution (Hoc, 1977). When using provider software, not only does the programmer require an understanding of the programming language used to write the client code, but also the

API resources, tasks and their syntax. We see programmers forming an understanding of what is to be implemented as a high-level understanding of the entities in terms of behavior and outcomes. Understanding how those entities are modeled in the API of the specific provider system and the syntax of the API methods is then used to integrate those entities to a program.

Therefore the understanding required to create programs using provider software requires understanding of the programming language used to write the client code and the parts of the overall program that do not use APIs with its syntax, grammar, rules of discourse and other conventions. It also requires understanding of the provider software and the API, which have their own design, syntax, rules of discourse, and other conventions — a mental model of the API and its language.

5. Conclusions

Writing about programming, Pennington and Grabowski stated "Programming problems are unique in that they usually involve solving a problem in another (application) problem domain, such as mathematics, accounting, electronics, or physics, in addition to solving the program design problem" (Pennington & Grabowski, 1990). Now, we can add a third level to the complexity of programming. Programming using provider software requires the programmer to solve the problem in the problem domain and solve the program design problem in terms of the structure and behavior of the program that solves the problem. It also requires the programmer to solve the problems of identifying and selecting provider software that provide the required resources to implement the designed program, and writing the client code to interact with the API of a provider software so that the provider software behaves as required.

6. References

Andrews, A., Ghosh, S., & Choi, E. M. (2002). A model for understanding software components. In *International conference on software maintenance, 2002. proceedings.* (pp. 359–368).

Atwood, M. E., Jeffries, R., Turner, A. A., Polson, P. G., & CO, S. A. I. E. (1980). The processes involved in designing software. *NTIS, SPRINGFIELD, VA, 1980, 62*.

Chen, X., He, J., Liu, Z., & Zhan, N. (2007). A model of component-based programming. In *International symposium on fundamentals of software engineering: International symposium, fsen 2007, tehran, iran, april 17-19, 2007. proceedings* (pp. 191–206).

Détienne, F., & Détienne, F. (2002a). Software reuse. *Software Design—Cognitive Aspects*, 43–55.

Détienne, F., & Détienne, F. (2002b). What is a computer program? *Software Design—Cognitive Aspects*, 13–20.

Heinonen, A., & Fagerholm, F. (2023). Understanding initial api comprehension. In *2023 ieee/acm 31st international conference on program comprehension (icpc)* (pp. 43–53).

Heinonen, A., Lehtelä, B., Hellas, A., & Fagerholm, F. (2023). Synthesizing research on programmers' mental models of programs, tasks and concepts—a systematic literature review. *Information and Software Technology*, 107300.

Hoc, J.-M. (1977). Role of mental representation in learning a programming language. *International Journal of Man-Machine Studies*, *9*(1), 87–105.

Kim, J., Lerch, F. J., & Simon, H. A. (1995). Internal representation and rule development in object-oriented design. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *2*(4), 357–390.

Mäkitalo, N., Taivalsaari, A., Kiviluoto, A., Mikkonen, T., & Capilla, R. (2020). On opportunistic software reuse. *Computing*, *102*, 2385–2408.

Marcella, R., Rowlands, H., & Baxter, G. (2013). The critical incident technique as a tool for gathering data as part of a qualitative study of information seeking behaviour. In *Leading issues in business research methods* (Vol. 2). Academic Conferences and Publishing.

Mosqueira-Rey, E., Alonso-Ríos, D., Moret-Bonillo, V., Fernández-Varela, I., & Álvarez-Estévez, D. (2018). A systematic approach to api usability: Taxonomy-derived criteria and a case study. *Information and Software Technology*, *97*, 46–63.

Pair, C. (1990). Programming, programming languages and programming methods. In *Psychology of programming* (pp. 9–19). Elsevier.

Pennington, N., & Grabowski, B. (1990). The tasks of programming. In *Psychology of programming* (pp. 45–62). Elsevier.

Thayer, K., Chasins, S. E., & Ko, A. J. (2021). A theory of robust api knowledge. *ACM Transactions on Computing Education (TOCE)*, *21*(1), 1–32.

Votipka, D., Rabin, S., Micinski, K., Foster, J. S., & Mazurek, M. L. (2020). An observational investigation of reverse engineers' processes. In *29th usenix security symposium (usenix security 20)* (pp. 1875–1892).