UTRECHT UNIVERSITY

MASTER THESIS

Simplifying the Complex: Strategies to reduce existing API Complexity using CPS techniques

Author:
Thomas NOLST TRENITÉ

Under supervision of:
Prof. Thomas KOSCH
Dr. Harry HALADJIAN

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in

Human Computer Interaction (HCI-5330963)
Department of Computing Science

June 14, 2023

Abstract

Software development plays a crucial role in driving technological advancements and fostering innovation across industries. However, the ever-increasing complexity in software systems poses significant challenges. This research aims to explore the origins of software complexity and develop strategies to mitigate it. Specifically, it investigates software complexity through the lens of Complex Problem Solving (CPS) and Software Comprehension, areas that have received limited attention thus far. The study focuses on API integration, a common area of software development, and examines two strategies (strategy A and B). These strategies incorporate the CPS techniques *decomposition* and *Lean Thinking* to address different dimensions of complexity inherent in API software. In a user study (N=30), participants had to retrieve data from an API platform and use it to build small conceptual prototype applications. Metrics related to time, successfulness in task and (perceived) effectiveness were utilized to assess the impact of the strategies on reducing interface and system complexity. The findings reveal how strategy A significantly reduces task completion times and the time required for the initial API call. Moreover, strategy A demonstrates superior performance compared to the baseline in metrics related to complexity. This highlights the effectiveness of concepts from Lean Thinking in reducing software complexity. Strategy B shows promising results in supporting developers' individual completion of API integration tasks and facilitating software comprehension by providing tailored learning materials suitable for common-sense learners. On the whole, developers expressed high levels of satisfaction with the effectiveness of both strategies in achieving their respective goals. This research establishes a foundation for enhanced interaction with complex software systems, opening avenues for future studies to explore more effective support strategies. The outcomes contribute to the advancement of software development practices and offer valuable insights for improving ease of use and reducing complexity in software systems.

Table of contents

Acronyms

API Application Programming Interface.

CES Customer Effort Score.
CPS Complex Problem Solving.

DevOps Development Operations.
DevX Developer Experience.

HCI Human Computer Interaction.
HTTP Hypertext Transfer Protocol.

IFT Information Foraging Theory.

JSON Javascript Object Notation.

MidFi Middle-Fidelity.

SDK Software Development Kit.
SRE Site Reliability Engineer.

UI User Interface.

1 Introduction

Software development has become an integral part of contemporary daily life, powering almost every large, complex system used in industries such as healthcare, finance and automotive. As software continues to play a crucial role in daily operations, the importance of being able to interact and understand software has become paramount. However, as software systems significantly increase in scale and size, so does the *complexity* involved. While the success rate of software projects depends on a range of aspects like time constraints or available resources, it is evidently influenced by the underlying, but often unnecessary complexity developers face while interacting with software systems (Raman, 1998, Schefer-Wenzl and Miladinovic, 2019). Although some complexity in software systems is considered essential, reducing avoidable complexity has become a critical issue, as it can avoid potential re-engineering efforts and reduce maintenance costs (Delange et al., 2015).

While there is a small amount of literature focusing on reducing software complexity (e.g., through model-based engineering) (Delange et al., 2015), limited research has yet considered the perspective of complex problem solving techniques. As such, this approach shows promise as an area for research (Schefer-Wenzl and Miladinovic, 2019). However, little is known about how complexity-reducing techniques can be effectively integrated through strategies present in the software development process, and what factors contribute to their effectiveness.

Against this background, the present research has put forward the following research question:

"How can complex problem-solving techniques be successfully integrated into strategies for reducing software complexity, and what factors contribute to their effectiveness?"

To be able to answer the research question, this research will describe the state of affairs through the use case of Application Programming Interface (*API*) integration. APIs are an essential component of modern software systems as they allow developers to access and use real-time data gathered by organizations to build new software applications. Although APIs offer packaged, reusable code functionalities to ease or enable the software development process, developers often experience complexity when interacting with this software, which makes APIs suited as a use case (Storey, Fracchia, and Müller, 1999). Additionally, existing techniques for complex problem solving have yet to be deployed for similar use cases. This research differentiates between *interface complexity* and *system complexity*, where interface complexity refers to complexity regarding overall ease of use and design of the *User Interface* (UI), and system complexity refers to the difficulty that arises when developers try to integrate and combine various pieces of data from APIs into their own software applications.

In order to delve into the relationship between software complexity and failure, this research paper performs a deep dive into problem solving and the *software comprehension process*. Through this approach, a distinction can be made between regular and complex problems. Additionally, the approach facilitates the discovery of how complexity affects the comprehension process of developers solving software related problems, such as integrating API data. The underlying premise behind this inquiry

is that by implementing support strategies to reduce complexity, a disruption within the software comprehension process can be restored, leading to an improved likelihood of successful outcomes.

The present study will employ a mixed-method approach to investigate the effectiveness of two proposed support strategies in reducing interface and system complexity. The methods will include both quantitative and qualitative data collection techniques, such as metrics based on time, task success, and perceived experience, as well as quotes and observations of general developer behavior. Results will be analyzed for significance using statistical analysis and visualizations.

This paper is structured as follows. Chapter 2 presents a comprehensive review of existing literature on complex problem solving (techniques), (early-phase) software comprehension, and the use case of APIs. Following the Design Process by Gibbons (Gibbons, 2016), this research will go through 3 iterative phases (see image 2.3). Chapter 3, the *Understand* phase, will involve conducting field research to validate and add findings from literature. By undertaking this approach, the aim is to gain a deeper understanding of how developers currently seek information, face complexity obstacles and learn to work with APIs, as research on these topics is limited (Gao et al., 2020). This aim will be accomplished by applying the *Information Foraging Theory* (IFT) (Pirolli and Card, 1999) to capture and declare developer behavior. Chapter 4 will cover the *Explore* phase, where highly effective support strategies for developers will be developed by uncovering not only developers' needs and preferences, but also how effective support should be represented, supported by brainstorming sessions with topic experts. *Middle-Fidelity* and *High-Fidelity* prototypes will be developed and iterated on through stakeholder feedback. Chapter 5, *Materialize*, will cover research hypotheses, data analysis, study design setup, and insightful results from end-user testing. Finally, Chapter 6 will provide a discussion of the study's most notable findings, limitations and recommendations for future work. The research will conclude by drawing conclusions and addressing the initial research question.

Contribution statement

The present research will contribute to the field of human-computer interaction (HCI) by gaining insights into how techniques for Complex Problem Solving can be implemented in strategies to reduce software complexity. The findings will provide insights into the early software comprehension process and assist developers in coping with software complexity.

Through answering the research question, this work makes three main contributions:

1. A new perspective on how CPS techniques can be used effectively in strategies to reduce software complexity.
2. A deeper understanding of the use case of APIs in software development and the challenges developers face in the process of integrating them.
3. Inclusive insights on the early software comprehension process for developers getting acquainted with an API, with specific emphasis on API learning.

2 Related work

Studying the scoped topics below has been the result of a detailed problem investigation carried out within the company TomTom (TomTom, 2022) beforehand. Since unnecessary complexity is currently a relevant issue for TomTom's future operations and performance, this research began by investigating the root cause of the problem. Thus, a deep dive has been made into literature mainly using Google Scholar (Google, 2022) and Scopus (Co, 2022) which are the most widely used literature search engine and research database respectively (LLC, 2022b, LLC, 2022a). Using keywords such as *API's*, *Developer Experience*, *Learning* and *Support Strategies* has made it possible to revert back to the problem origin and to interpret it from different perspectives.

Based on the literature review, three topics have emerged related to the origin of the problem: *Complex Problem Solving*, *software comprehension* and facets of *learning*. Their relevance to the problem will be clarified by investigating them separately. Thereafter an argumentation is provided for connecting these topics together, highlighting the clear gap this research fills and distinguishing it from existing research. The upcoming sections will highlight the most relevant findings from literature, serving as a base for this research.

2.1 Complex Problem Solving (CPS)

2.1.1 Problem Solving

The cognitive process of human problem solving is identified as one of the main basic life functions within the brain (Wang and Chiew, 2010). Commonly accepted is the fact that the human brain makes approximately 35000 decisions per day, most of which are related to solving distinct problems (Lamata, Pelta, and Verdegay, 2021). Problems consist of three main elements: givens (available information), goals (desired termination state of solution) and operations (actions executed to achieve solution goals) (Elis Ormrod, 1999, Wang and Chiew, 2010).

Problem solving has become a main topic of interest in the last couple of years. As industries are showing a strong shift towards software development, there is an exponentially increasing need for developers with problem solving skills. Software developers solve problems by repeatedly formulating (on-demand) goals and objectives as they seek and find new information needed to accomplish their tasks (Bradley, Fritz, and Holmes, 2022). They seek this information through various useful information sources such as technical documentations and -blogs, which account for roughly $\frac{1}{3}$rd of the time spent on a problem (Li et al., 2013).

2.1.2 Problem Complexity

Most problems humans come across are well-defined, meaning that a goal definition can be extracted easily and it is clear how to progress towards this goal to reach a solution (Wang and Chiew, 2010). However, finding a solution to a problem might not even exist within the solver's *solution space* and this could be attributed to the

fact that the problem is ill-defined, or *complex*. In contrast to well-defined problems, complex problems have no clear goal definition and make it unclear how to allow progress towards a solution (Schefer-Wenzl and Miladinovic, 2019, Dörner and Funke, 2017, Foshay and Kirkley, 2003). Consequently, complex problems are in direct relation with failure and lack of progression which can lead to several effects such as trying to escape the situation or violation of rules (Dörner, 1980). Because of these effects, it becomes clear why the ability to succeed in complex problem solving is one of the most requested professional skills from employers (Schefer-Wenzl and Miladinovic, 2020 Sánchez Carracedo et al., 2018).

The field of CPS offers many possible research directions. The current study is scoped down to software developers trying to solve complex software problems.

2.1.3 Dealing with Complex Software Problems

Dealing with complex software problems can be very hard as developers struggle to extract clear goals and pathways, as discussed in subsection 2.1.2. Since there are many ways to possibly deal with such problems, the upcoming subsections will highlight two of the most prominent techniques for dealing with complex problems. Note that general problem-solving techniques are not discussed in this work, because these do not focus on creating more comprehensible goals and can therefore not be compared to techniques for CPS. However, this does not imply that similarities in solution approaches are non-existent between general problem-solving techniques and techniques for CPS.

Decomposition

Seen as a cornerstone for computational thinking, the most basic technique for tackling complex problems is *decomposition*. Decomposition, breaking down the complex problem into a set of smaller subproblems, has been proposed by a vast amount of research studies (Gick, 1986, Saenz and De Russis, 2022, Rijke et al., 2018). This process can repeat itself until the problem solver's scope is limited enough and has an understanding of the problem and a solid solution focus (Robins, Rountree, and Rountree, 2003). In the context of problem solving, novices may find the recursive process of extracting subproblems challenging and applying them even more difficult. In contrast, experts apply (recursive) decomposition with more ease, perform generation and evaluation of alternative solutions and are able to apply a known solution to newly found subproblems (Gick, 1986).

Value concept (from Lean Thinking)

Lean Thinking, commonly used in regular process development cycles in the manufacturing industry, emphasizes the importance of identifying and delivering the highest value to users while eliminating waste (Thangarajoo and Smith, 2015). This is achieved by focusing on user demand instead of adopting every possible feature users may want into their system, a practice that can result in unnecessarily complex software. Hence, Raman, 1998 argues how Lean Thinking can be applied to software development. Additionally, this way of dealing with complex problems has proven effective in a variety of industries, reducing cost, improving quality and increasing customer satisfaction.

Lean Thinking consists of several concepts, including the key concept *'value'* and its derivative, the *'value stream'*.[1] Value can be defined as anything that is perceived as value by the customer. First and foremost is to understand clearly what the user needs and what they consider as value. Next, the value stream comprises the set of actions required to bring value to the user. Every piece of value that is included in the stream helps eliminate existing activities that do not contribute any value, strengthening the value stream. In this paper, these two concepts will be applied in practice.

2.1.4 Learning to Solve Complex Problems

The importance of teaching and learning CPS is made clear by the quickly rising amount of technical courses offered to students in their curricula. Research conducted by Schefer-Wenzl and Miladinovic, 2019 utilized Bloom's Taxonomy (Bloom, College, and Examiners, 1964) to visualize specific levels of knowledge. For each level, authors defined clear tasks for students to fulfil. Tasks on each level are then addressed by using course methods such as gamification, videos, learning diaries and anchored instructions. The final learning stage concerns the application of knowledge into a real software integration project.

Further research done by Foshay and Kirkley, 2003 has suggested the use of 17 principles focusing on teaching problem solving, of which two are relevant for CPS. One, *'encouraging learners to use knowledge in defining a goal'* has been applied by the research from Schefer-Wenzl and Miladinovic, 2019, further confirming its validity. Another principle supports the use of inductive teaching strategies to tackle complex problems as these *'encourage synthesis of mental models'*. Although this teaching method could support CPS, there is no recent research putting this into practice. Due to the general shortage in the available amount of research on posing methods or concepts for teaching and learning CPS (Schefer-Wenzl and Miladinovic, 2019 Van Deursen et al., 2003), it is encouraged to adopt educational approaches into solutions aimed at addressing complex problems.

2.1.5 Differences in Learning Styles

The concept of learning styles, although according to Mian et al., 2022 appears to have become fragmented in literature over the past decade, remains significant to consider as individuals tend to differ in their approach to learning something new. In the context of solving complex software problems, it is crucial to keep in mind that developers may differ in their preferred learning approach. For instance, Coffield et al., 2004 describes how the 4MAT model is widely adopted to improve teaching methods. This model identifies different types of learners, including *imaginative*, *analytic*, *common-sense* and *dynamic* learners. Other studies however describe four general dimensions using the Index of Learning Styles (ILS), with each dimension having two opposing categories (Felder, Soloman, et al., 2000). Moreover, there exist several learning strategies that can be utilized to control the memory process and gain concentration (Mian et al., 2022), which appear to be consistent with characteristics of learning styles discussed in Coffield et al., 2004. Additionally, Mian et al., 2022 argues that incorporating different evaluation methods, such as rating scales and questionnaires, can provide learners with valuable insights into their own learning styles and problem-solving approaches.

[1] Readers interested in learning more can refer to the source for a more comprehensive overview.

2.2 Software Comprehension

As been made clear in the preceding section, there exists a clear need, especially by developers, to find support in solving complex software problems. As software complexity increases and goals are harder to define, developers try to create more manageable structures for themselves in order to better comprehend the process. Research conducted by Klemola and Rilling, 2002 has highlighted that assessing the comprehension efforts of a software developer helps in determining effective support approaches, such as a methodology or tool. Software comprehension can be described as *"a process whereby a software practitioner understands a software artefact using both knowledge of the domain and/or semantic and syntactic knowledge, to build a mental model of its relation to the situation"* (O'brien, 2003).

For this reason, the upcoming subsections aim to understand these efforts by unveiling the software comprehension process, approaches, and known mechanisms to facilitate it.

2.2.1 Mental models / process

In order to understand software related concepts, developers make use of cognitive processes and information structures to form a so-called *mental model* (Seel, Al-Diban, and Blumschein, 2000, Storey, Fracchia, and Müller, 1999). In the context of software development, a mental model refers to the developer's mental representation of the software to be understood. In general, the more developed a mental model is, the better understanding an individual will have of a task or problem and therefore the solution towards it. A developer constructs various mental models (e.g., domain, program) and is able to switch between these during the comprehension process (Storey, Fracchia, and Müller, 1999).

In order to better understand the software comprehension process (see figure 2.1), the *knowledge-based understanding model* is widely accepted and frequently used (O'brien, 2003, Storey, Fracchia, and Müller, 1999, Letovsky, 1987). The model consists of three interdependent components: (1) *knowledge base*, (2) *mental model* and (3) *assimilation process*. The software comprehension process is a cycle starting at the knowledge base, which consists of concepts such as domain knowledge, plans and goals. The knowledge base then stimulates the assimilation process, which is a process that helps to construct the mental model by processing the acquired knowledge. Although this theory is still agreed upon, recent research conducted by Kadar et al., 2021 added a sub-component of knowledge base, namely *external representation*. This sub-component covers all materials supporting programmers in their software comprehension process (e.g., *developer documentation*, *source code*, *tutorials*).

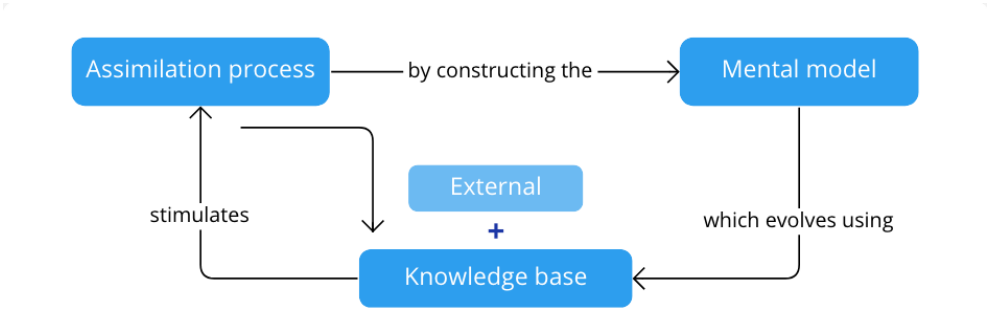

FIGURE 2.1: Software comprehension process

With this information in mind, the origin of the described need can now be better narrated: developers are trying to solve a complex problem, but very often fail because they cannot specify program goals needed to map to the relevant parts of the solution implementation (a disruption of the assimilation process). As a consequence, the mental model cannot evolve properly and becomes underdeveloped, resulting in a lack of global understanding of the program, indirectly causing failure when dealing with complex problems.

2.2.2 Facilitating Software Comprehension

Based on the approaches above, there exist a variety of support mechanisms to facilitate the comprehension process. The most known technique, *reverse engineering* (Storey, Fracchia, and Müller, 1999), is a technique assisting the developer in creating high-level abstractions from source code which is typically done by deconstructing individual components. While many reverse engineering tools exist, the practice of reverse engineering is often associated with malicious purposes and is known for being time-consuming (Klemola and Rilling, 2002). These factors can significantly impact solution quality, particularly when solving complex problems.

More assistance techniques are rising in usage and education, such as *systematic code reading* or technologies for visualizing software in a certain way (Hebig et al., 2020). However, dedicated research has not yet provided a definitive answer regarding the most efficient techniques for facilitating software comprehension, particularly for novices. An appropriate starting point would be to develop comprehension techniques that lay essence on providing the right information at the right time, as recent research found that this helps novices *"answer comprehension questions with significantly more accuracy, in less time"* (Adeli et al., 2020).

2.3 Use Case: API Integration

Developers frequently encounter failure when attempting to solve complex problems during the software development process. This observation is justifiable, given that *"the majority of software engineering and integration problems is of complex nature"* (Schefer-Wenzl and Miladinovic, 2019). One of the most common use cases regarding software integration is undertaking the process of integrating an API (Jonnada and Joy, 2019). In order to provide a comprehensive understanding of what APIs are and why they are relevant to employ for a case study, the upcoming subsection will present a technical background on APIs.

2.3.1 Technical Background

APIs, short for Application Programming Interfaces, are sets of protocols, routines, and tools for building software applications. They serve as an intermediary service between different software applications, allowing them to communicate and exchange data. Think of APIs as a messenger that helps different software applications talk to each other by sharing data, just like how a waiter communicates the ordered food to the kitchen, and also delivers the food to customers.

The present research focuses specifically on Web APIs, serving as a bridge between different software services offered by companies over the web in exchange for compensation. Web APIs enable developers to access and retrieve data that companies make publicly available, allowing them to develop custom software solutions. For

instance, APIs can be used when building a webshop that needs to connect to a payment gateway to enable customers to pay for their orders securely. Another example is the Twitter Search API allowing developers to provide methods for developers to interact with Twitter Search and trends data (Twitter, 2023b). Commonly, use cases for APIs include application integrations, but also web apps, dashboards or mobile applications (Postman, 2020).

Adoption

According to a report from ProgrammableWeb.com, the number of web APIs has increased substantially over the last 10-15 years (elaborated by Vaccari et al., 2021, see figure 2.2). The report reveals that in 2022, the number of web APIs was estimated to be around 24,000 and this number continues to change. A reason for the rapid growth in API use is the constantly increasing demand for data, with organizations being supported in creating new APIs. As a result, the market for publicly offered software has been expanding. (Myers and Stylos, 2020).

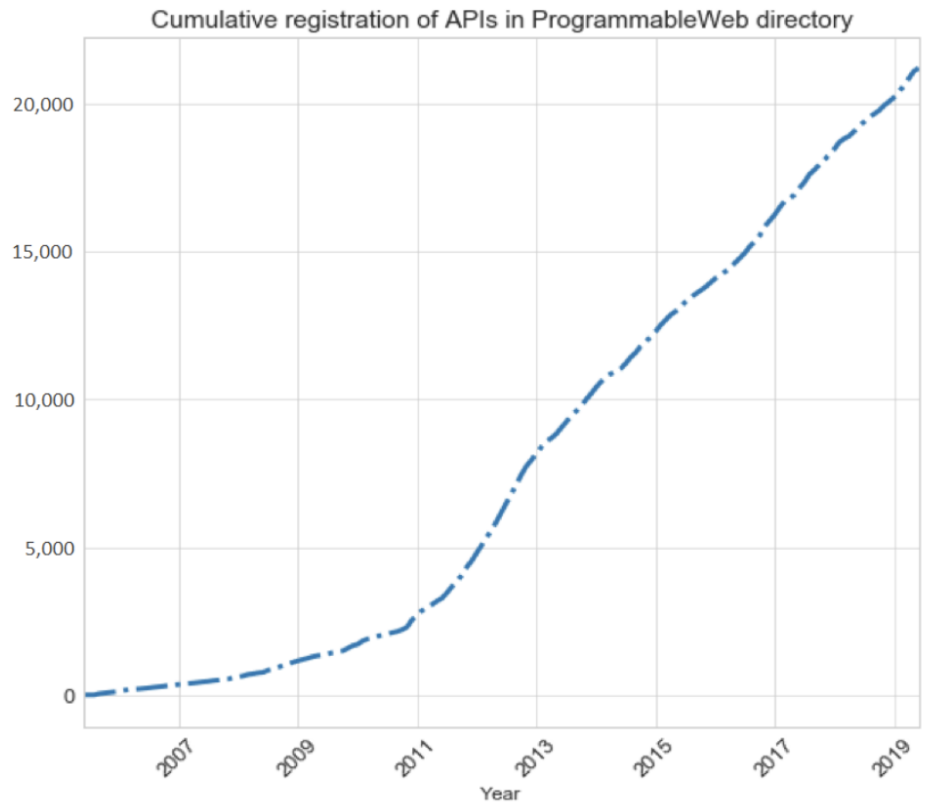

FIGURE 2.2: Adoption of web APIs since 2005 (cumulative) (Vaccari et al., 2021)

Design

The way an API is designed highly impacts its performance. A good API design requires a lot of expertise, research and resources to implement and is thus of high value to be considered early in the process, before development commences (Postman, 2020). It is about choosing what information to display and what not to display, as some information might not be needed to be understood for good use of

the service. There exist metrics for measuring API usability such as functional efficiency, overall correctness and learnability for novices (Myers and Stylos, 2020). When designing APIs, it is common practice to measure derivatives of these metrics by applying heuristic evaluation guidelines defined by large consulting firms such as the Nielsen Norman Group (Nielsen, 1994).

Documentation

To assist developers in being able to understand how communication with APIs work, companies typically offer *Developer Documentation*. This documentation type consists of every possible resource developers might need in order to be able to work with an API. Think of video tutorials, reference documentation, code samples and forums. In order to provide a web API of high quality, a company needs to offer developer documentation that is clear, comprehensive and easy to understand.

Communication

APIs communicate using different types of standard HTTP requests such as *GET* and *POST*. A GET Request is used to retrieve data from a server. When a developer makes a GET request to an API, it sends a request to the server, asking for specific data such as products or a user profile. The server usually responds with the requested data in a specific format, such as *JSON* (JavaScript Object Notation). Furthermore, a developer can create POST requests, allowing for the submission of data to a server in order to create a specific resource such as performing an order to create a new user account. While other types of API requests are covered in greater detail in existing literature, the present research will provide an example of a GET request.

Suppose a developer wants to employ the Twitter API to integrate Twitter data into his own application (Twitter, 2023a). He likes to obtain a list of the most trending topics at the moment. By making a GET Request, the developer is able to receive this data. The base of the GET Request would be:

```
GET https://api.twitter.com/1.1/trends/place.json
```

This request will return a collection of trend objects, however they will be empty as no specific id is specified. Therefore, the request can be updated by adding certain *parameters*, which are additional pieces of information. In this use case, the parameter `id` represents the location from where to return trending information. Suppose `id` 1 represents global trending topics. In that case, the request would take the following form:

```
GET https://api.twitter.com/1.1/trends/place.json?id=1
```

The server will now respond with the corresponding data the developer asked for, in the form of a JSON object containing the trend-objects that the developer is after. A sample response is included in Appendix A.

To conclude

APIs play a vital part of modern software development. Without the use of APIs, applications and systems could not work together seamlessly. However, APIs can become complex due to the infinite amount of use cases, overloads of information

and evolving standards. Therefore, it is of high essence to find solutions to reduce this complexity, keeping APIs powerful and easy to use.

2.4 Unaddressed Solution Opportunities

A recap on complex problem solving, software comprehension and APIs had resulted in various directions to perform scientific research on, some of which still not considered yet. Even though there is a reasonable understanding on how to deal with complex software problems, application of techniques seems to be scarce. Although software comprehension is closely related to CPS, studies still display a relatively small focus on this topic (Van Deursen et al., 2003, Hebig et al., 2020) which is unjustified as this topic is key to program understanding and creating high-level solutions. Recently developed techniques, such as linkable annotations (Adeli et al., 2020), can provide valuable insights for facilitating the comprehension process. However, these techniques may lack contextual information, making it difficult to capture the broader context in which software elements operate. This context is crucial for fully comprehending the software system as a whole.

While it is desirable to conduct additional research on facilitating software comprehension, there is greater value in taking a broader perspective and focusing on a specific phase of software comprehension. Forming a correctly developed mental model starts at the early phase of the software comprehension process, where the developer has not yet extracted any value out of the system. As empirical research on this phase is still underrepresented (Hebig et al., 2020), this study will put emphasis on facilitating the early comprehension phase. Theories on early information seeking by developers can help acquire insights within this specific area. Moreover, a review of existing literature revealed a gap in empirical research concerning the CPS perspective on facilitating the (early) software comprehension process. Addressing this research gap can lead to the establishment of highly effective technology.

2.4.1 Solution Opportunities for APIs

In the absence of adequate support, developers are required to identify and put together the necessary knowledge to achieve a well-functioning assimilation process.

The process, especially for complex problems like API integration, can be excessively time-consuming and result in failures and reduced performance. Concentrating on early comprehension, an optimal solution would denote a developer's ability to comprehend and integrate certain parts of a complex API by making use of existent support. This may entail the development of a small conceptual prototype or working through a 'getting started' project to ensure effortless integration at a later comprehension phase.

In the upcoming chapters 3 to 5, the design process towards developing support strategies for developers in reducing software complexity will be presented. This process follows the stages of understanding the problem, exploring potential solutions, and materializing the final solution, as proposed by Gibbons (Gibbons, 2016)

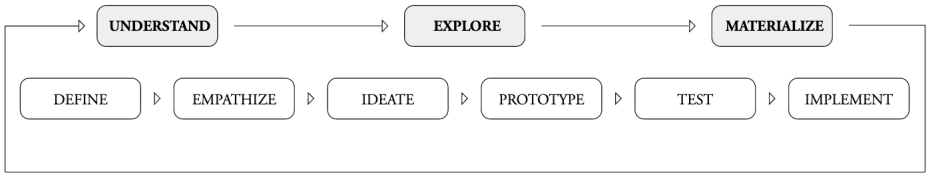

FIGURE 2.3: Design Process by Gibbons (Gibbons, 2016)

3 Understand

The related work and technical background presented the latest research findings regarding problem complexity, software comprehension and their implications for APIs. The Understand phase consists of two components, namely *Emphasize* and *Define*, which aim to gain a practical understanding of how the theories presented in Chapter 2 are manifested. Moreover, this chapter will derive an understanding of developers' needs for adequate support. Through interviews that observe developers' behaviors, these observations will be refined into specific research objectives that should be taken into account during the later stages of the design process.

3.1 Preliminary Interviews

Preliminary interviews were performed with two groups of stakeholders. The first group concerned API-familiar software engineers working at TomTom. This group is familiar with the APIs as they possess a deeper level of knowledge and experience by simply using TomTom APIs in their daily work, therefore having a better understanding of its practices and overall design. The second group consisted of TomTom personnel who possess technical knowledge related to APIs, either due to their background or job titles indirectly associated with APIs. This group, in this research referred to as *API-unacquainted*, are individuals who are either unfamiliar with the TomTom APIs or have not built *acquaintance* yet. Consequently, they lack basic awareness of how the API services offered by TomTom function. Reaching acquaintance can be achieved through activities such as reading the documentation or learning about the features, design and structures in any other way.

The primary goal for these interviews was to confirm complexity related bottlenecks experienced by developers when interacting with APIs to build software. By using this approach, solutions to problems can be discovered that are supported by concepts from existing literature.

The interviews were conducted following a structured script that included a set of predefined questions and prompts to guide the conversation (see Appendix H). Each interview lasted approximately 40 minutes and was audio-recorded for later reference. Detailed notes were taken during the interviews to capture key points and observations. The recorded interviews were transcribed verbatim to ensure accuracy in data analysis and interpretation.

Conducting interviews with appropriate stakeholders using the *think-aloud* method led to increased clarity on various topics, including API usability issues, learning and integration. The think-aloud method involves participants verbalizing their thoughts while performing a task, providing insights into their cognitive processes. To maintain a clear research focus, only insights are discussed that fall within the defined scope. For instance, these include the most important information sources for developers in the software comprehension process and how API complexity affects a developer's mental model. Subsequently, similar insights were clustered using *affinity mapping*. Affinity mapping is a technique used to categorize and organize

ideas or information into meaningful clusters, leading to the identification of distinct, newly discovered themes. In the upcoming section, these insights and their corresponding themes will be explored in greater depth.

3.1.1 Unacquainted group

The unacquainted group consisted of a DevOps manager, a Site Reliability Engineer (SRE) and a newly joined Apigee[1] expert. These interviews were focused on foraging behaviour of (un)acquainted developers interacting with the TomTom Web APIs. The APIs covered in these sessions were *Traffic*, *Map Display* and *Routing* and respectively provide developers with access to real-time traffic information, maps, directions and route optimizations.

Among the insights, the following were most notable for the scope of this research:

- Unacquainted developers looking to work with a new API often think *use-case first*. With a particular use case in mind, developers strategically navigate to the locations they perceive as potential sources of assistance. Their thoughts and feelings are centered around a use case that suits their specific purpose.
- Unacquainted developers are taking their time to comprehend the information presented to them and are actively seeking clarification by asking questions about any aspects that appear unclear. As such, findings from the literature study on software comprehension can be confirmed: in order to form a mental model of a software related concept, developers are using information structures and are actively seeking cues to feed their domain knowledge and assimilation process.
- Developers find value in both a *getting started* overview and specific information within the developer documentation. Related work (section 2.4) has demonstrated how these insights can be declared by means of theories on early information seeking by API-unacquainted developers. The following text will delve into one of the most well-established and supported theories in this regard.

IFT Applied to TomTom APIs

To further explore the perceived value of different information sources by API-unacquainted developers, the lens of Information Foraging Theory (IFT[2]) can be applied (Pirolli and Card, 1999). This theoretical framework clarifies how developers actively seek out information and adapt their search strategies based on the expected utility of information they encounter, which is crucial to predict what information an unacquainted developer would need and when. These insights can be used to develop support strategies that provide the right information at the right time.

Insights reveal that, in foraging terms, the most 'profitable' sources of information contain a *get started overview*, *examples* (formats, use cases), *input parameters*, and *end points*[3] (often curl or URL commands). Despite developers having different information needs and interests changing over time, generally speaking these are the most

[1] Apigee is a platform for developing and managing APIs

[2] IFT has been used in related developer studies (Fleming et al., 2013, Sedhain and Kuttal, 2022, Lawrance et al., 2010) and is officially accepted as a valid and useful theory in the field of HCI and psychology (Olson and Olson, 2003).

[3] When an API interacts with another system, the touchpoints of this communication are considered endpoints.

important information sources developers need when trying to implement the API data into their application. Repetitive pathways taken by developers in their information foraging process also substantiate these findings.

As predicted by IFT, switching between the profitable sources is done rapidly to minimize search time, but often look at the same source more than once to try and achieve an efficient *rate of gain*. According to the unacquainted group, finding the information needed for a successful integration is not the challenge in common API developer documentations. Instead, it's more about the time it takes to find this information, mainly due to a high foraging cost (information overload and complexity), leading to a suffering rate of gain. The results give rise to a research objective focused on simplifying foraging API information:

Research objective 1
Evidence-based practice must react accordingly to the current foraging behavior of developers interacting with unfamiliar APIs by reducing existing foraging costs through the rebuilding of the *value stream* and delivering the most profitable information sources (*value*) at the right time.

3.1.2 API-Familiar group

In-person interviews were held with five TomTom developers who had an average of 9 years of programming experience with various TomTom APIs. Each developer had built familiarity with diverse APIs such as *Map Display*, *Reverse Geocoding*, *Routing* and *Waypoint optimization*. These interviews provided further insights into the variations in complexity levels across different API domains. The script of this interview can be found in Appendix H. Below, a summary is presented of the most valuable insights obtained.

- **Familiarity and Acquaintance** - Becoming acquainted with an API (i.e., forming a basic awareness of what it does and how it works) can be reached in a relatively short timeframe of one hour or less (depending on the developer's previous experience). However, reaching familiarity (i.e., a deeper level of understanding) can be seen as more of a constant, ongoing process. The speed to getting familiar can vary depending on the complexity of the API and the developer's level of expertise. According to reports from developers in the API-familiar group, reaching familiarity with a new set of APIs typically takes between one to three months.
- **Comprehension** - API-familiar developers tend to adopt a *top-down* approach when reading the content of the Developer Documentation. A top-down approach involves scanning the documentation, incorporating example requests or code fragments into their application, and executing them. If necessary, they revisit the documentation to refine their solution. This iterative process enables them to update their mental model. Interestingly, the API-unfamiliar group showed similar behavior, suggesting a comparable approach to advancing their understanding. This can be attributed to their existing knowledge of APIs in general.
- **Complexity** - Complexity in APIs can manifest in two different forms, which arise at different stages of the developer's interaction with APIs. To visualize the findings related to the nature of complexity in software development, a Developer Journey was created and can be found in Appendix D.1. The journey provides a broader context for the study, outlining how developers typically *Discover*, *Evaluate* and *Register* (for) the API system, before progressing through

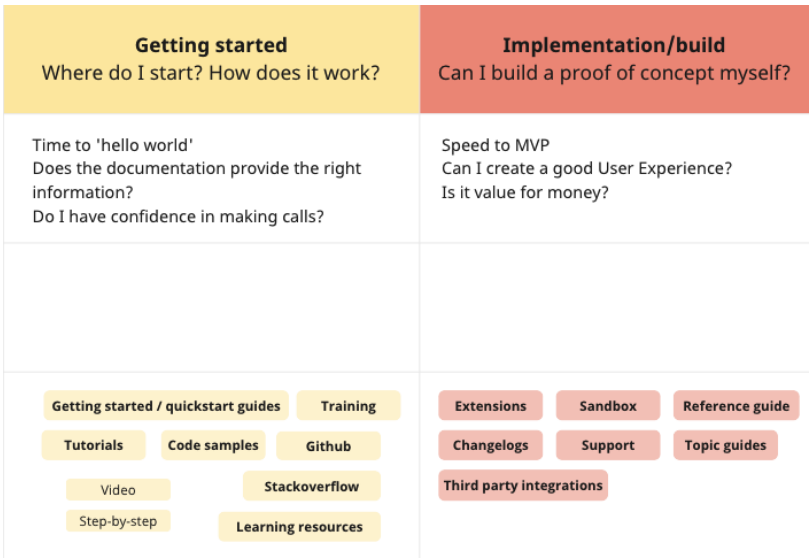

FIGURE 3.1: Two phases of the Developer Journey hold different complexity types

the *Getting started* and *Implementation* phases shown in Figure 3.1. These latter two stages are the focus of this research since it was found that two types of complexity emerge here and will now be differentiated.

On the one hand, developers are constantly dealing with *interface complexity* in their *getting started* phase. This type of complexity refers to how difficult it is for a developer to understand existing content from the API, such as its structures, methods and parameters. In practice, developers are mainly faced with interface complexity when having to deal with an overload of parameters and unclear documentation, leading to errors and inefficiencies.

On the other hand, *system complexity*, which relates to the interconnections between API information leading to composition issues, seems to be present once developers take the data from the API and implement it into their own prototypes. To integrate an API into a new or existing application, developers frequently compose APIs by combining parameters, functions, requests, and URLs, which are necessary to retrieve the desired data. This composition often involves incorporating existing code from SDKs[4] or other APIs. As an example, consider a developer tasked with creating a Taxi Cars Dispatcher app using the services from TomTom. In order to build this application, a developer would need to know how to create a TomTom map, create and customize markers using the SDK, how to work with the Routing API to calculate the shortest route and how to draw the route from the taxi to the location of the passenger. Hence, it is not surprising that combining multiple API elements to solve a problem is seen as one of the main barriers to API learning (Kelleher and Brachman, 2023).

[4] An SDK, or Software Development Kit, is a set of tools and resources that enables developers to create applications and interact with APIs more easily by providing pre-built functions and interfaces.

3.1.3 Lacks of existing support

In line with the insights presented above, it is crucial for organizations providing API services to acknowledge the importance of providing adequate support to developers in their processes. While tutorials seem to be the 'golden standard' for aiding developers in their early comprehension process, they may fall short in reducing software complexity, as they often adopt a passive approach to learning instead of actively engaging developers in working with complex software.

Research objective 2
To empower developers in effectively engaging with complex software, it is crucial to offer support that considers the learning curve and provides a holistic understanding of the software. This support should not be constrained in silos for developers to make sense of independently, but should actively guide them through the learning process to facilitate a well-functioning software comprehension process.

4 Explore

The primary goals of the *Explore* phase are to investigate the needs of developers working with complex software and explore ideas that can address those needs identified in the *Understand* phase. The Explore phase consists of two steps: *Ideate* and *Prototype*. During this phase, brainstorming sessions with stakeholders will be conducted to generate a wide range of ideas, which will then be filtered to ensure that the tested solutions in the *Prototype* step are realistic, feasible and of the highest impact for solving the challenge at hand.

4.1 Mapping Developer Needs

Although a part of the interviews in the previous chapter (*Understand*) have been conducted with a developer group familiar with TomTom APIs, the upcoming phases will prioritize the unacquainted group of developers. The needs of this group are more severe and therefore a larger potential could be unlocked.

To identify the specific requirements that future solutions must meet, brainstorm sessions were held with internal stakeholders such as API Architects and Product Managers from the *Developer Experience* (DevX) department. Involving these stakeholders in the ideation process will not only lead to solution-approaches that are more interdisciplinary and up-to-date with the latest knowledge in the API and Developer Experience fields, but also ensure that the approaches are feasible implementation wise. One example is API architects who have a realistic view on limitations that may affect developers and think of solution-approaches that fit in the current API design. On the other hand, DevX experts will think of solution approaches that keep in mind the already available tools and resources and difficulties that arise in those particular efforts. This knowledge contributes to a greater solution scope and therefore increases the chances of finding effective ways to manage API Complexity.

4.1.1 Brainstorm session 1

```
CONFIDENTIAL: This content includes confidential information belonging
   to TomTom, and as such, it cannot be disclosed or shown to the
   public. This information is proprietary and protected under non-
   disclosure agreements.
```

4.1.2 Brainstorm session 2

```
CONFIDENTIAL: This content includes confidential information belonging
   to TomTom, and as such, it cannot be disclosed or shown to the
   public. This information is proprietary and protected under non-
   disclosure agreements.
```

4.2 Strategy filtering

4.2.1 DVF Framework

4.2.2 Supplementary Filters

4.2.3 Strategy A

4.2.4 Transition of Strategy B

4.3 Prototyping

4.3.1 Prioritizing characteristics

4.3.2 MidFi Prototyping

MidFi prototype evaluation

CONFIDENTIAL: This content includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

4.3.3 HiFi Prototyping

CONFIDENTIAL: This content includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

Technology - Strategy A

CONFIDENTIAL: This content includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

Technology - Strategy B

CONFIDENTIAL: This content includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

Piloting

CONFIDENTIAL: This content includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

5 Materialize

The Materialize phase marks the final stage of the design process and fulfills the objective of this study to identify effective support strategies for reducing API complexity. To comply with this objective, testing with the initial target audience of end-users will be covered in the current phase. However, it is crucial to acknowledge that the successful implementation of the identified support strategies is dependent on various external factors beyond the scope of this research. Since this research was conducted at TomTom, the implementation of these strategies is dependent on the company's available resources and decision-making processes. Thus, this study does not aim to provide definitive insights into the level of implementation of these strategies in the end-product. Nevertheless, plausible hypotheses based on the current understanding will be formulated and tested with the target audience through a usability study in the current phase.

5.1 Data and Analysis

This study utilizes several key metrics or variables to effectively determine study outcomes. A clear distinction has been made between *independent*, *dependent*, and *confounding* variables. The screen recordings enable the accurate collection and filling in of these metrics by tracking relevant data points.

Task Difficulty & Support Strategy

To observe the effect of the dependent variables, two main independent variables were identified, also known as the *predictor variables*. As previously mentioned, the tasks given to participants will be manipulated by increasing their difficulty, making task difficulty an independent variable. Additionally, the specific support strategy provided to participants will be varied, making it another independent variable.

Several dependent variables were defined and observed or measured in response to changes in the independent variables. This will enable the establishment of cause-and-effect relationships between both variables. The dependent variables contain three overarching themes: *time*, *Successfulness in Task* and *(perceived) Experience*. Reasoning for these variables will be provided shortly, and their contribution to the value of this research will be argued for.

Task Completion Time

Measuring the time to complete a task appears to be a viable approach in evaluating the effectiveness of support strategies. It helps determine if developers can achieve greater accuracy and efficiency in task completion, as discussed in Section 2.2.2. This is particularly relevant when integrating API data into an application, as this process often requires a significant duration. Furthermore, it provides an indication of the overall usability of the strategies and how they perform in that regard. If developers can work easily with the strategies, *task completion times* may improve compared

to baseline, thus helping to prove that the support strategy improves overall user experience for developers.

Time to First Successful API Call

Secondly, the *Time to First Successful API Call*, also known as *time to 'Hello world'*, was considered. This metric facilitates the measurement of how quickly developers are able to find the right information at the right time and get started with help of strategy A. A faster initial API call serves as a reliable measure of developers' ability to quickly start and can indicate the increased effectiveness of strategy A in reducing interface complexity.

Number of requests (attempts) developers make until they achieve a successful call - number of reach outs to external sources

To account for differences in comprehension time and preferred learning styles, a complementary metric to the Time to First Successful API Call was used, namely the *number of requests (attempts) developers make until they achieve a successful call*. Utilizing this approach allows for evaluation of the effectiveness of strategy A in assisting developers to make the correct call in fewer attempts, even if the Time to First Successful API Call exceeds the average. Another crucial variable considered is the *number of reach outs to external sources* made by developers. Although reaching out to external sources for help is a common practice among developers, this metric helps to clarify whether (for example) *strategy B* offers the right sort of descriptions to satisfy developers' information needs, as well as identify opportunities for improving the existing work.

Task Success - Customer Effort Score

In the present study, one of the most prominent metrics measured is *Task Success*. This metric requires additional context to ensure proper understanding. While completing tasks, participants were observed by an expert in the field, who understands how to solve the tasks at hand. When a participant got stuck or was lost in a task, the expert could steer the participant in the right direction by providing subtle *nudges*. It should be noted that these nudges are only provided in situations where the expert expects a developer to be unable to complete a task or when a significant amount of time has elapsed without any progress. By providing nudges, the expert never fully reveals the exact next step for a developer or provides developers with any concrete instructions on how to proceed, therefore causing as less influence as possible on final outcomes.

Task Success is measured using a 0 - 0.5 - 1 scale, where a score of 1 indicates successful completion of the task with little difficulty and without any use of nudges, 0.5 indicates successful completion through using a few nudges, and 0 indicates wrongful completion or an incompleteness of the task even when nudges are provided. The Task Success metric provides valuable insights into the amount of assistance needed to complete tasks with and without the use of proposed support strategies, which can be calculated and compared directly.

In addition to these themes, two ratings-based variables were incorporated - the *Customer Effort Score* (CES) and *Self-estimated REST API usage score*, both of which have been discussed earlier. CES, as defined by Qualtrics, 2021 and Clark and Bryan, 2013, reflects the customer's perception of the amount of time and energy that they

have to spend to get an issue resolved, a request fulfilled, a product purchased/returned or a question answered. This study applies CES by using it to measure the effort-related aspect of working with complex APIs and investigate the effect of the support strategies on perceived effort.

Self-estimated REST API usage score

Alongside CES, the *Self-estimated REST API usage score* has also been included to overlook the distribution of experience between participants. This metric is essential as it helps to determine the degree to which API experience affects overall performance, and whether the support strategies have led to an improvement in overall performance or only benefited more experienced participants.

Programming Experience - state of Developer Portal

Finally, two confounding variables were identified. Although not the primary focus of this study, it is imperative to acknowledge their potential impact on the outcomes to ensure a thorough analysis of this study's results. Two confounding variables that could impact the study outcomes are *varying levels of experience in programming*, specifically in JavaScript, and *the current state of the Developer Portal* including its documentation.

5.2 Hypotheses

This study aims to investigate if CPS techniques can be successfully integrated in strategies for reducing interface and system complexity in software. To test these research questions, hypotheses were formulated. First, two general null hypotheses were created to test the absence of an effect:

H0- 1 *The integration of Lean Thinking through strategy A does not result in a significant reduction in interface complexity compared to when no support is provided.*

H0- 2 *The integration of decomposition through strategy B does not result in a significant reduction in system complexity compared to when no support is provided.*

Next, to evaluate the effectiveness of both strategies and to test expected effects of various factors on their effectiveness, a set of general hypotheses was formulated. These hypotheses are based on the metrics defined in section 5.1 and have been clustered into previously mentioned themes: *Time*, *Successfulness in task*, and *(Perceived) Experience*.

H1 - Hypotheses Concerning Time

H1.1 *Strategy B* significantly reduces task completion times compared to baseline during API data retrieval and integration

H1.2 *Strategy A* significantly reduces task completion times compared to baseline during API data retrieval.

H1.3 *Strategy A* significantly reduces time to first successful API call compared to baseline during data retrieval.

H2 - Hypotheses Concerning Task Successfulness

H2.1 *Strategy B* will increase task success scores compared to the Baseline.

H2.2 *Strategy B* will significantly reduce the number of reachouts to external sources compared to the Baseline.

H2.3 *Strategy A* will significantly reduce the number of erroneous requests made by developers before achieving a successful call compared to the baseline.

H2.4 *Strategy B* will significantly reduce the number of erroneous requests made by developers before achieving a successful call compared to the baseline.

H2.5 *Strategy B* will show a consistent increase in comprehension ratings over tasks compared to the baseline

H2.6 *Strategy A* will show a consistent increase in comprehension ratings over tasks compared to the baseline

H3 - Hypotheses Concerning Perceived Experience

H3.1 *Strategy A* will lead to less required effort to make the right call using the TomTom documentation compared to the Baseline.

H3.2 *Strategy B* will lead to less required effort to build a working prototype using the data from the API compared to the Baseline.

H3.3 *Strategy A* will lead to increased confidence and perceived ease of use compared to the Baseline.

H3.4 *Strategy B* will lead to increased confidence and perceived ease of use compared to the Baseline.

H3.5 *Strategy A* will lead to increased perceived effectiveness in finding the right information at the right time compared to the Baseline.

H3.6 *Strategy B* will lead to increased perceived effectiveness in building a working prototype compared to the Baseline.

5.3 Study Design

A qualitative study was run in which developers' behaviour and measurements were collected while using APIs with which they had neither built acquaintance nor familiarity. The present study employs a between-subject design, where performance of the baseline (BL) is compared to *Strategy A* (SA) and *Strategy B* (SB), and participant groups will test each task while being randomly assigned and exposed to one of these conditions. The setup of this study will now be discussed in detail below.

5.3.1 Pilot

In total, six participants were recruited (two per condition), with ages ranging from 18 to 32 years old (M = 25.6, SD = 4.24). The pilot was used to make final modifications on the template code and instructions.

5.3.2 Participants

A total of 31 participants were included in the study, with eleven assigned to the baseline condition, nine in the *SB* condition, and eleven in the SA condition. For the baseline and *SB* condition, this was achieved through convenience and snowball sampling in combination with sending invitations through a mailing list of a coding school. For the SA condition, participants were recruited through UserTesting (UserTesting, 2022). The baseline condition had ages ranging from 18 to 30+ (M = 26), strategy B had ages ranging from 18 to 30 (M = 25) and an average age of 31 was found for the participants using strategy A.

Most of the 20 participants in the baseline and *SB* conditions were students (N=9) or full-time workers (N=8), while the minority of participants worked part-time (N=3). Contrary, all eleven participants in the *SA* condition were pursuing full-time positions. As for education levels, the baseline condition had participants with predominantly high school education, followed by bachelor's and master's degrees. In the *SB* condition, most participants had a bachelor's degree, followed by high school and master's degrees. In the *SA* condition, most participants had a master's degree, followed by a bachelor's degree and high school education.

To meet the study's target audience, participants were required to have a neutral to strongly positive experience with using a REST API and have little to no prior experience using the TomTom APIs. Additionally, they were required to be currently pursuing or recently completed a degree in Computer Science or another field where programming is a common practice. The mean programming experience was 7 years for baseline and *Strategy A* (Med = 6 and 7, respectively) and 6 years for *Strategy B* (Med = 5).

Participants were requested to provide self-estimates of their programming experience and REST API experience on a Likert scale ranging from 1 to 5[1]. The mean self-estimated programming experience scores were 3 for the baseline condition (Med = 3), 3 for the *SB* condition (Med = 3), and 4 for the *SA* condition (Med = 4). The self-estimated scores for REST API experience were comparable across all conditions.

5.3.3 Methodology

Materials

After recruiting participants who met the criteria for the target audience, each session started by preparing the according environments and setting up questionnaires. In order to commence with a session, participants needed a laptop or got one provided by TomTom. On this laptop, a folder was included which contained the following materials for the test session:

- An API key
- A Visual Studio Code[2] environment where the tasks could be located via separate folders. For each task, a local server was set up before the testing took place.
- A document containing individual links to all questionnaires.
- A task instruction file
- A link to the Developer Portal of TomTom

[1] 1 = no experience, 5 = highly experienced

[2] Visual Studio Code is a text editor and development environment.

Procedure & Tasks

First, participants were presented with a consent form (see Appendix E), which they were required to read and sign. In accordance with the protocol, participants were compensated with a 15,- bol.com gift card for their time and effort. Following agreement to the written consent form, each participant was requested to fill in an introductory demographic survey (Q0) (see Appendices F.1 and F.2). As discussed previously, this survey answered questions concerning age, level of education, employment status, amount of programming years, self-estimated programming experience and self-estimated REST API experience.

Next, participants were informed that they would be experimenting with the TomTom APIs by (a) retrieving data and (b) utilizing it to develop their own conceptual prototype application. Participants were instructed to complete as many of the three tasks as they could and complete tasks using their natural approach. As developers normally do not think out loud during programming, the *Think Aloud method* was discarded. The test setup ensured that participants were in a distraction-free environment to perform the tasks to ensure the accuracy of findings. Participants were also made aware of the option to reach out to external sources, such as the internet, if they preferred.

To be able to make a thorough analysis of participants' behaviour and to effectively measure data, screen recording software was used to record the screens of participants during the session.

As previously mentioned, participants were observed by an expert in the field during task completion. Consequently, when participants encountered significant obstacles or became disoriented, the expert occasionally offered nudges which could impact the Task Success Score. In addition to providing nudges when necessary, the expert was also available for technical questions or clarifications related to instructions. These technical questions and clarifications were not considered as nudges in the analysis of this study.

As part of this study, participants were asked to complete a set of API-programming tasks within a one-hour timeframe. This approach was based on the findings of Gao et al., 2020, which suggest that including tasks can boost API learning. This is particularly relevant, as increased knowledge about the API should lead to improved comprehension of its features and functionality.

According to Kelleher and Brachman, 2023, tasks should be involved around combining multiple API elements to solve a problem. Therefore, tasks in the experiment have been set up in such a way that it requires a participant to use the Developer Portal (and thus *strategy A*) to request data from an API connected to a use case, and then use the front-end environment (and thus *strategy B*) to integrate data by displaying it onto the map. According to Holmqvist and Jungermann, 2021, dividing tasks per use case help to define a clear goal definition which is needed to enhance the software comprehension process. To accurately measure comprehension effects by applying learned concepts on the way, tasks were created purposefully to increase in difficulty and divided into subtasks (a to c) (Holmqvist and Jungermann, 2021). This approach ensured that participants completed the full hour and provided maximum insights for this part of the study.

To maintain realistic usage scenarios and comply to the inheritance of storytelling principles defined in subsection 4.1.1, participants were provided with a storyline in

which they were planning a day out in Amsterdam and wanted to be well-prepared by accessing real-time data on locations, traffic, and routes. Therefore, the following tasks were set up:

- In **Task 1** participants were requested to display a marker on the map using the coordinates of Amsterdam, retrieved through the Developer Portal.
- **Task 2** posed a higher level of difficulty by requiring participants to extract real-time data on traffic incidents from the API and display an instance of a closed road on a map by adding a marker to it. This task increased the level of difficulty by (a) providing a less complete template, (b) enforcing more detailed JavaScript knowledge and (c) having to work with more specific parameters in the Developer Documentation.
- In **Task 3**, participants were instructed to visualize a route from point A to B. This task not only required participants to work with complex JavaScript logic but also interact with the *Routing API*. As the preliminary interviews pointed out, the Routing API was considered the most complex API due to its extensive interface complexity and information overload. For these reasons, this task was designed to be the most challenging of all.

After participants completed a task, their perceived effort was assessed to determine the Customer Effort Score (CES) in this study. In Q1 to Q3 (see Appendices F.3 to F.6), which consisted of the same set of questions, participants were asked to rate the difficulty or ease of performing a specific action. In the SA condition, the action involved making the correct API call using the TomTom documentation, while for the SB condition, the action was to build a functional prototype using the data from the API. It should be noted that, in compliance with research guidelines established by TomTom, the Likert Scale used in this survey ranged from 1-7, with 1 indicating *Very difficult* and 7 indicating *Very easy*.

Additionally, after task completion, in Q1 to Q3 participants were requested to rate their level of agreement on the following statements to assess their comprehension: *"The environment provided me with the right information to make the API call"* (SA) and *"The environment helped me accomplish the programming task"* (SB).

After the one hour timeframe finished, participants were debriefed and kindly requested to fill in Q4 (see Appendices F.7 to **??**), where a request was made to reflect on the overall complexity and effectiveness of the strategies.

- The complexity of the task was measured by asking participants how *confident* they felt in using the platform and tools effectively, as well as assessing their *perceived difficulty* in understanding the API system as a whole.
- Based on the assigned condition, participants were asked for their opinion on the effectiveness of a specific strategy in accomplishing a particular goal. For SA, the goal was to 'find the right information at the right time', while for SB the goal was to 'build a working prototype'. For this study, it is crucial to not only obtain a qualitative understanding of the effectiveness of the strategy, but also the rationale behind the given ratings. Since all questions for participants so far were based on a rating scale of 1-5 or 1-7, a follow-up question was included to ask for clarification on participants' rating of effectiveness.

At the end of the experiment, participants were given the opportunity to provide their general feedback, such as justification for their behavior or any noteworthy observations during the experiment, whether positive or negative.

5.4 Results

5.4.1 Quantitative

In this section, results of the data collection will be presented and analyzed. Next to means (M) and standard deviations (SD), data was tested on normality using the *Shapiro-Wilk test* and significance level of 0.05 before any statistical testing was performed.

To determine the most suited method(s) for statistical analysis, it is essential to be reminded of the exact stage in the Developer Journey where the strategies are present. The Developer Journey shows that both strategies, although complementary, suit different purposes and solve different types of API complexity. Therefore, *strategy A* and *strategy B* are not being compared to each other. To compare both strategies separately to the baseline condition, a *two-sample independent T-test* will be employed, which is suited for time-related data. However, this method cannot be applied to data such as Likert scales as they are often ordinal. That is, while values have a natural order, the differences between the values may not be equal. Consequently, to apply statistical tests to the interval data, the *Mann-Whitney U test* will be utilized to compare mean interval values of both conditions compared to the baseline. This statistical test is commonly used to compare differences between two independent samples and is therefore appropriate for non-parametric data.

This analysis will now focus on the results derived from the independent metrics on the basis of Bar Charts. It should be noted that due to an insufficient amount of data points for Task 3 in conditions SB and BL, findings on Task 3 will occasionally be excluded from any further analysis.

Task Completion Time

```
CONFIDENTIAL: This content includes confidential information belonging
    to TomTom, and as such, it cannot be disclosed or shown to the
    public. This information is proprietary and protected under non-
    disclosure agreements.
```

Time to First Successful API Call

```
CONFIDENTIAL: This content includes confidential information belonging
    to TomTom, and as such, it cannot be disclosed or shown to the
    public. This information is proprietary and protected under non-
    disclosure agreements.
```

Number of erroneous requests (attempts) developers make until they achieve a successful call

```
CONFIDENTIAL: This content includes confidential information belonging
    to TomTom, and as such, it cannot be disclosed or shown to the
    public. This information is proprietary and protected under non-
    disclosure agreements.
```

Number of reach outs to external sources

Task Success Score

Customer Effort Score

Comprehension

Complexity

Support Effectiveness

5.4.2 Qualitative

Although used metrics allow for a thorough analysis of quantitative data, earlier conducted phases in this study have led to support strategies substantiated by various qualitative inputs. As mentioned previously, it is of importance not only to qualitatively understand strategy effectiveness, but also reasoning behind developers' actions during the sessions to further support these measures. Consequently, quantitative Developer behaviour was measured during and after conducting end user testing sessions. This measurement involved keeping track of their foraging behaviour, comments, and support calls. Additionally, short follow-up conversations were conducted right after each session to address general feedback and opinions

on both support strategies. Topics that contribute most value to the existing findings in this study will now be addressed.

Foraging Behaviour

Outcomes of held experiments show consequent findings with literature from the related work section. When learning to work with and build acquaintance with a new API, developers in this study consistently showed a process of bottom-up comprehension. This is done by inspecting small building blocks of information in the API design to gradually build up to a comprehensive understanding. Additionally, it was found that creating well-designed tasks provides valuable direction and steering to simulate the comprehension process, as they drive developers to seek profitable information sources and experiment with different integrational logic APIs offer. Moreover, it was discovered that proficient REST API developers show the perception that they already possess the necessary knowledge to locate the required information, occasionally leading to them spending more time looking for the right location of the information. After a sufficient amount of time had elapsed, developers were rapid in adjusting their approach to locate the right information.

Demand Characteristics

End user testing sessions have shown how developers tend to perceive tasks as easy when nudges are applied by an expert in the session. This phenomenon is known as *Demand characteristics* (part of 'moderator effects') and may influence the participants' behaviour and responses. Referring back to this study, it could be the case that nudges might have influenced developers' perceived effort in completing tasks, measured by the CES. This experiment has tried to minimize the influence demand characteristics have on affecting study outcomes by providing subtle nudges that were nonspecific and handed out only in certain circumstances, as can be read in section 5.3.3.

Preferred Learning Style

```
CONFIDENTIAL: This content includes confidential information belonging
    to TomTom, and as such, it cannot be disclosed or shown to the
    public. This information is proprietary and protected under non-
    disclosure agreements.
```

Lacks in current API Design

```
CONFIDENTIAL: This content includes confidential information belonging
    to TomTom, and as such, it cannot be disclosed or shown to the
    public. This information is proprietary and protected under non-
    disclosure agreements.
```

6 Discussion

Results from this study reveal the overall effectiveness of support strategies in solving different types of complexity in software systems, applied to the context of API integration. The following section will interpret the study results through literature and the main research question, to assess if initial hypotheses can be confirmed or remain unproven. It is worth mentioning that the focus will be on the most impactful results and corresponding hypotheses, as these are most relevant to this research. This approach aims to provide a clear and comprehensive discussion.

6.1 The Interplay of Time and Support in Programming Tasks

Task Completion Times for the *SB* condition were consequently higher compared to when no support was provided. Although no statistical difference between Task Completion Times for BL (baseline) and SB (strategy B) was found (hence not confirming H1.1), a comparison with Task Success Scores reveals an interesting insight. It is relevant to mention that, although it is beneficial for a developer to be able to integrate data into his own application quickly, *strategy B* was not designed to increase the integration time. Instead, its purpose is to assure developers are able to individually work through the complexity by going through a step-by-step process to complete the task at hand. The latter was measured using the Task Success Score, which makes it meaningful to compare results from both these metrics. One noteworthy finding is that although *strategy B* showed longer task completion times, the Task Success Scores consistently showed a corresponding increase. This means that although (on average) developers take a few minutes longer to complete a task using *strategy B*, fewer nudges are needed to successfully complete the task at hand using a natural approach. Although statistical testing could not prove any significance between conditions, a moderate effect size indicates how these findings are fairly reliable and show support for H2.1. Moreover, quantitative feedback from developers highlighted how *strategy B* can support developers in gaining clearance for the next step when needed, possibly explaining the increase in success scores between BL and SB.

Further analysis revealed a significant difference between Task Completion Times for *strategy A* and Baseline. *Strategy A*, initially designed to provide developers with the right information at the right time, demonstrated its value by reducing task completion times by an average of 63.7% across all three tasks. Even though *strategy A* was tested through a clickable prototype, the extent of its impact was substantial and participants did not differ much in terms of skill and experience, which further increases the reliability of this finding. The analysis presented above confirms the validity of H1.2. A reasonably coherent decrease in Time to First Successful API Call between *strategy A* and Baseline was also found and proved to be statistically significant, thus providing further support for the confirmation of H1.3. The reliability of this insight is further enhanced by the inclusion of qualitative data: a notable rating of 4.7 out of 5 in terms of perceived effectiveness indicates a high level of developer satisfaction with the strategy.

Based on the analysis, it is evident that by only displaying specific information that is most needed in their 'getting started' phase (in Lean Thinking known as *value*), *strategy A* can be considered very effective in reducing existing interface complexity. Additionally, these findings dictate how *strategy A* can be a robust strategy for helping novices in their early phase answer comprehension questions with significantly more accuracy, in less time. The upcoming subsection will examine whether the direct comprehension ratings align with these findings.

6.2 Key Factors Affecting Comprehension: An Overview of Influential Elements

Theory from the literature review has illustrated how adequate support can help developers in their early comprehension process by empowering the assimilation process to form a well-developed mental model of unfamiliar, complex software (Kadar et al., 2021, Adeli et al., 2020). This section will interpret results focused on software comprehension to find out if proposed support strategies are effective in providing needed comprehension assistance for developers.

6.2.1 Learning effects and Task difficulty influence comprehension effects

When no support is provided, developers tend to exhibit higher levels of agreement on the API environment providing the data needed for completing the task at hand between Task 1 and Task 2. Learning effects could be the main reason for this: although the API for Task 1 and 2 varied, the data retrieval method was consistent. By identifying patterns and commonalities in Task 1, developers can update their mental model with more knowledge about where to make the API call. These informed decisions result in developers becoming more efficient in their search for information in Task 2, hence declaring the increase in comprehension effects.

Although Q1 showed increased ratings in the baseline condition, the comprehension ratings for Q2 slightly decreased. This observation can be further explained by the *expertise reversal effect* (Kalyuga, 2007). A study by Gupta and Zheng, 2020 found that novices lacked the adequate mental model to comprehend the complex content in their learning materials when attempting to solve a task for the first time. Therefore, additional support was provided through text explanations to aid comprehension. However, as novices gained more knowledge, the additional text information became unnecessary for their learning, which demonstrates the expertise reversal effect. This effect may have contributed to the lower comprehension ratings shown in the results, as developers may tend to rely more on open-ended problem solving approaches rather than a step-by-step approach to solving a problem.

Moreover, individual differences in programming or JavaScript experience may have played a role in these findings as developers who had less experience in these areas tended to struggle more when working on tasks with increased difficulty. According to Klemola and Rilling, 2002, developers with less general programming experience and domain expertise may have slower comprehension, which could have contributed to the decrease in comprehension ratings observed between tasks.

6.2.2 Learning styles and over-reliance affect comprehension

Contrary to H2.5, the analysis of *strategy B* condition revealed a significant and therefore conspicuous **decrease** in comprehension ratings from Task 1 to Task 2 compared to the baseline. This finding suggests that developers faced greater challenges as task difficulty increased. While there are factors that could lead to an increase in comprehension, such as identifying commonalities in used code templates or using generic examples to acquire contextual knowledge, it is important to consider the substantial decrease in comprehension effects. This decrease could potentially be attributed to an *over-reliance* on support.

Especially when still API-unfamiliar, developers might have become overly dependent on the support provided in Task 1. When developers have to deal with increased task difficulty in Task 2, they rely on *strategy B* to find answers. However, *strategy B* requires developers to apply their knowledge and skills in a flexible way by using generic explanations and -examples. Although this was not particularly challenging in Task 1, qualitative inquiries highlight instances where developers experienced confusion in Task 2. This confusion arose when *strategy B* presented generic examples instead of functional ones, and it can be attributed to a developer's preferred way of learning. As sources from related work predicted (Coffield et al., 2004, Mian et al., 2022), these findings show how learning styles can indeed vary among developers and that *strategy B* might not always offer the most suitable learning style for purely practical (by Coffield et al., 2004 defined as *common-sense*) learners, occasionally leading to decreased comprehension effects over tasks.

Although the data did not align with the anticipated results for H2.5, *strategy B* was rated 3.7 out of 5 and received overall positive feedback. For developers trying to understand basic principles and concepts of unknown software systems, *strategy B* has shown promise as a strategy. By offering theory and examples that focus on core concepts and functionality, *strategy B* strikes a balance between providing a pathway and goal to developers in their early comprehension phase, while trying to keep developers encouraged to actively engage in the process of comprehending software. Nevertheless, future research is needed to determine the effects of functional examples on developers' reliance on support and comprehension processes.

6.2.3 Inconvenience issue leads to inconsistent comprehension ratings

Comprehension ratings have shown to fluctuate between tasks completed in the *SA* condition. Further examination revealed how developers faced an inconvenience issue while completing Task 2 in the Figma prototype, which is likely the reason for the analyzed inconsistency. Although this study found a statistically significant decrease in comprehension ratings in Task 2 between BL and SA, the reliability of this is up for debate. Additionally, it is challenging to determine whether comprehension ratings would be more consistent if a fully functional prototype of *strategy A* would have been set up, however analysis still showed an increase in comprehension ratings from Task 1 to Task 3, indicating that *strategy A* might contribute to increased software comprehension effects for developers retrieving data from complex APIs. However, comprehension ratings for Task 1 and 3 between BL and SA did not indicate a significant difference, presenting inconsistencies with H2.6.

6.2.4 How *Strategy A* facilitates Information Foraging Process

A substantial increase in comprehension ratings between Task 1 and Task 3 was observed. Additionally, the analysis of developer behavior from recordings revealed that developers naturally search for information sources that provide them with the most value in completing the task at hand. For this, they are using their existing knowledge on APIs gathered through general experience, enriched by tasks completed earlier in the session.

Upon task completion, most developers reach out to the same location in the developer documentation for the new task and proceed with the same steps towards solution as in the task before, only to deviate when not being able to find the data required for the new task. This foraging behaviour is in line with insights from IFT that argue: when the value of the information is right, users stick to that information to further enhance their comprehension process (Lawrance et al., 2010). If they are not finding valuable information in a particular section of the environment, they may switch to a different source within the environment in search of better *information scent* (Fleming et al., 2013).

Consequently, these results elicit the importance of dealing with existing interface complexity, especially within the Developer Documentation as this is where API Calls can often be made. *Strategy A* has shown to facilitate developers in finding the most profitable information sources with more accuracy, in less time. By inheriting value and value stream as concepts from Lean Thinking, *strategy A* is able to answer comprehension questions with more accuracy by saving valuable time for developers (Adeli et al., 2020).

Overall, it remains difficult to determine whether proposed support strategies contribute to a restored software comprehension process. Initial interviews conducted with API-familiar developers have revealed that becoming familiar with an API can take several weeks or even months, depending on personal expertise and the level of complexity involved. This observation is supported by the literature, which indicates that developing a fully coherent mental model is a time-consuming effort (O'brien, 2003) and that several months may pass before direct changes can be observed (Seel, Al-Diban, and Blumschein, 2000).

The present research has focused solely on the direct effects of comprehension, and these may not be sufficient to draw conclusions on the overall impact of support strategies on the software comprehension process. Although the support strategies developed in this study aim to assist developers in enhancing their software comprehension processes, future research should consider measuring long-term comprehension effects when interacting with complexity in APIs to ensure the most reliable findings (Balijepally, Nerur, and Mahapatra, 2015).

6.3 Coping with Complexity

As shown by literature, coping with increased complexity in software systems such as APIs can be challenging. This subsection will analyze the results regarding complexity, as measured by confidence levels and perceived ease of use, by revisiting literature-defined techniques for Complex Problem Solving (CPS). The aim is to determine the effectiveness of the CPS techniques embedded in *strategy A* and *strategy B* in reducing complexity.

6.3.1 *Strategy A* increases overall confidence and ease of use

Analysis shows how *strategy A* outperforms the baseline in terms of confidence and perceived ease of use by developers. A significant difference was found between SA and BL for overall confidence, where *strategy A* leads to more overall confidence in using the system compared to when no support is provided. It can be argued that this is because of *self-efficacy* (Bouffard-Bouchard, 1990, Moores and Chang, 2009): When support is provided by means of *strategy A*, developers may feel more confident in using the system and achieving their goals overall as they have access to resources which they know helps them overcome interface complexity they otherwise would have to deal with. This increased sense of self-efficacy is likely the reason for the increase in confidence. In direct relationship with the measured confidence level, a significant difference was found between SA and BL for perceived ease of use. This significant difference serves as an additional layer of confidence in stating that *strategy A* is effective in reducing existing interface complexity. The findings above provide support for H3.3.

More research is needed to find out if (more concepts from) Lean Thinking can be applied to more strategies and use cases for reducing (interface) complexity. However, with respect to API integration, implementing concepts from Lean Thinking in a strategy has proven to be very effective for reducing interface complexity.

6.3.2 Subtasks help developers create main goal and pathways towards solution

Our analysis did not reveal significant differences between *strategy B* and baseline for both confidence and perceived ease of use, thus not confirming H3.4. Therefore, it can not be conclusively stated that inheriting decomposition in a strategy leads to a decrease in system complexity. Nevertheless, findings suggest that *strategy B* appears promising in guiding developers to establish a clear goal definition, as evidenced by slightly increased confidence levels and qualitative feedback. This confirms findings by Holmqvist and Jungermann, 2021. The way in which *strategy B* is structured, with its use of subtasks, suggests a positive outlook towards facilitating a valid assimilation process for developers. By visualizing a pathway towards a goal, *strategy B* is able to assist in the assimilation process within early software comprehension. These findings are consistent with theories presented by Foshay and Kirkley, 2003 and Schefer-Wenzl and Miladinovic, 2019.

Overall, analyzing the results has demonstrated how *strategy A* outperforms the baseline significantly with respect to time, ease of use, and confidence levels. Therefore, H0-1 was rejected. While *strategy B* exhibited higher task success scores motivated by effect size and qualitative feedback, this study did not find any statistically significant differences between its performance and that of the baseline. As a result, H0-2 could not be rejected.

6.4 Limitations, Constraints and Future Work

This study poses limitations that may have affected the outcomes of the study design approach as well as the generalizability of results.

First, this study has been limited to the utilization of Web APIs with code templates being built using JavaScript. As a result, results obtained from this study may not

be applicable to other API types and programming languages on which they are developed, highlighting the need for further research to explore the applicability of findings in different contexts. Additionally, despite efforts to screen participants with experience in (API) programming and familiarity with JavaScript, individual differences in expertise levels among developers were still plausible, potentially impacting the study results.

Based on preliminary interviews, this study has categorized API complexity into interface and system complexity, both found present in different stages of the Developer Journey. While the presence of both complexity types was evident in end user testing sessions, preliminary interviews were limited in depth and size and may not have captured the full range of complexities involved in working with APIs in real-world settings. More comprehensive research is needed to fully capture and account for all possible types of complexity developers encounter when interacting with complex APIs.

The study's design poses two main limitations. First, the *Hawthorne effect* (Sedgwick and Greenwood, 2015) could have influenced participants' behaviour and performance as they completed programming tasks in the presence of a topic expert, which would have potentially led to inaccurate results. Although participants were assured confidentiality and comparisons between participants were not disclosed, the impact of the Hawthorne effect on the study results cannot be entirely ruled out. Secondly, demand characteristics may have affected study outcome validity as task success occasionally relied on subtle nudges. While the study design accounted for this through the Task Success Score, the subjectivity of what counts as a 'subtle nudge' could differ from person to person, which could introduce study result bias. Although nudges were necessary to accurately measure support strategy impact, they were not predetermined but instead given at repetitive moments. Hence, their potential impact on success scores cannot be ruled out completely. We encourage future work to investigate alternative ways of measuring the impact of nudges and reducing their potential influence on success scores.

One main constraint of this study is the use of a clickable prototype of *strategy A*. Due to a lack of available resources, a fully functional version of the developer portal could not be developed. Although extensive work has been put into creating interaction flows within the clickable prototype, it still did not include all necessary features of a 1:1 copy, interfering with the validity of results related to time or usability. Future work should put more time and effort into developing a fully functional prototype. A second constraint of this study relates to the limited sample size. Due to the difficulty of finding participants suited for this study, the amount of data collected for each condition was relatively small. As a result of the small sample size, participants were not screened on learning style which has been shown to impact the perceived effectiveness of support strategies. Future work should aim to increase the sample size to improve the validity and reliability of the study results.

The limited timeframe of this research project has been a notable constraint. While it is possible that immediate comprehension effects are stimulated by proposed strategies, fully developing a mental model of software takes more time than the duration of conducted tests allow. Therefore, a follow-up longitudinal study measuring comprehension over 1-3 months is needed to accurately assess the impact of proposed support strategies in enhancing software comprehension.

In conclusion, future work should aim to discover more effective strategies to address the complexity in software systems beyond those examined in this study. Findings of this study have revealed how incorporating CPS techniques in support strategies is a viable method to develop such strategies. Considering the vast amount of CPS techniques still untapped, there is significant potential for further exploration. Moreover, the study only examined support strategies in the context of API integration, and further research is needed to determine their effectiveness in other domains.

To further increase the potential of *strategy B*, future work could also point directions towards exploring machine learning and AI techniques for automating the step-by-step process to decompose complexity in software systems.

In summary, this study has produced promising findings regarding the effectiveness of applying techniques from Complex Problem Solving in strategies to reduce software complexity. Moreover, it has identified influential factors to consider for future explorations. Keeping the limitations in mind, there is ample room for further investigation in the field of Developer Experience, offering valuable prospects for optimizing the utilization of APIs and other complex software on a global scale.

7 Conclusion and Outlook

This study has looked into the performance of two distinct support strategies in reducing existing software complexity, applied in the context of API integration. By following a structured design process supported by related work, *strategy A* and *strategy B* were evaluated on various performance metrics.

The present research found that *strategy A* significantly reduces task completion times and time to first successful API call. *Strategy B* has shown to reduce the amount of assistance developers need from a topic expert in completing tasks. Although no (positive) significant differences were found between *Strategy B* and baseline, qualitative inquiries show positive feedback towards decomposing a complex problem into subtasks to create a pathway towards a solution for developers.

While the definitive impact of this approach on the software comprehension process should be measured in a longitudinal study, results indicate promising opportunities for improving short-term comprehension effects.

To conclude, decomposition and Lean Thinking are two techniques for CPS that can successfully be inherited in strategies for reducing software complexity (in the context of API integration). However, this area of research is still in its infancy as there is still a great deal of work to be done to uncover more effective developer support for complex software problems. As we continue to navigate the challenges of increasingly complex software systems, the demand for new approaches has never been more pressing. This research marks the first steps towards that goal.

Bibliography

Adeli, Marjan et al. (2020). "Supporting code comprehension via annotations: Right information at the right time and place". In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 1–10.

Balijepally, VenuGopal, Sridhar Nerur, and RadhaKanta Mahapatra (2015). "Task mental model and software developers' performance: An experimental investigation". In: *Communications of the Association for Information Systems* 36.1, p. 4.

Bloom, Benjamin Samuel, Committee of College, and University Examiners (1964). *Taxonomy of educational objectives*. Vol. 2. Longmans, Green New York.

Bouffard-Bouchard, Therese (1990). "Influence of self-efficacy on performance in a cognitive task". In: *The journal of social Psychology* 130.3, pp. 353–363.

Bradley, Nick C, Thomas Fritz, and Reid Holmes (2022). "Sources of software development task friction". In: *Empirical Software Engineering* 27.7, pp. 1–34.

Clark, Moira and Andrew Bryan (2013). "Customer effort: help or hype?" In: *Henley Business school*.

Co, Elsevier (2022). *Scopus*. https://www.scopus.com.

Coffield, Frank et al. (2004). "Learning styles and pedagogy in post-16 learning: A systematic and critical review". In.

Delange, Julien et al. (2015). "Evaluating and mitigating the impact of complexity in software models". In.

Dörner, Dietrich (1980). "On the difficulties people have in dealing with complexity". In: *Simulation & Games* 11.1, pp. 87–106.

Dörner, Dietrich and Joachim Funke (2017). "Complex problem solving: What it is and what it is not". In: *Frontiers in psychology* 8, p. 1153.

Elis Ormrod, Jeanne (1999). *Human Learning.(3e edition). Upper Saddle River (NJ): Merrill*.

Felder, Richard M, Barbara A Soloman, et al. (2000). *Learning styles and strategies*.

Fleming, Scott D et al. (2013). "An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.2, pp. 1–41.

Foshay, Rob and Jamie Kirkley (2003). "Principles for teaching problem solving". In: *Technical paper* 4.1, pp. 1–16.

Gao, Gao et al. (2020). "Exploring programmers' api learning processes: Collecting web resources as external memory". In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 1–10.

Gibbons, Sarah (2016). *Design Thinking 101Google Scholar*. https://www.nngroup.com/articles/design-thinking/.

Gick, Mary L (1986). "Problem-solving strategies". In: *Educational psychologist* 21.1-2, pp. 99–120.

Google (2022). *Google Scholar*. https://scholar.google.nl.

Gupta, Udita and Robert Z Zheng (2020). "Cognitive Load in Solving Mathematics Problems: Validating the Role of Motivation and the Interaction among Prior Knowledge, Worked Examples, and Task Difficulty." In: *European Journal of STEM Education* 5.1, p. 5.

Hebig, Regina et al. (2020). "How do Students Experience and Judge Software Comprehension Techniques?" In: *Proceedings of the 28th International Conference on Program Comprehension*, pp. 425–435.

Holmqvist, Axel and David Jungermann (2021). "Optimizing the usability of REST API reference documentation". In.

Jonnada, Srikanth and Jothis K Joy (2019). "Measure your API complexity and reliability". In: *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, pp. 104–109.

Kadar, Rozita et al. (2021). "Program Comprehension Technique in Teaching and Leaning: A Cognitive Perspective". In.

Kalyuga, Slava (2007). "Expertise reversal effect and its implications for learner-tailored instruction". In: *Educational psychology review* 19, pp. 509–539.

Kelleher, Caitlin and Michelle Brachman (2023). "A sensemaking analysis of API learning using React". In: *Journal of Computer Languages* 74, p. 101189.

Klemola, Tuomas and Juergen Rilling (2002). "Modeling comprehension processes in software development". In: *Proceedings First IEEE International Conference on Cognitive Informatics*. IEEE, pp. 329–336.

Lamata, Maria T, David A Pelta, and José Luis Verdegay (2021). "The role of the context in decision and optimization problems". In: *Fuzzy Approaches for Soft Computing and Approximate Reasoning: Theories and Applications*. Springer, pp. 75–84.

Lawrance, Joseph et al. (2010). "How programmers debug, revisited: An information foraging theory perspective". In: *IEEE Transactions on Software Engineering* 39.2, pp. 197–215.

Letovsky, Stanley (1987). "Cognitive processes in program comprehension". In: *Journal of Systems and software* 7.4, pp. 325–339.

Li, Hongwei et al. (2013). "What help do developers seek, when and how?" In: *2013 20th working conference on reverse engineering (WCRE)*. IEEE, pp. 142–151.

LLC, Paperpile (2022a). *The top list of academic research databases*. https://paperpile.com/g/academic-search-engines/.

– (2022b). *The top list of academic search engines*. https://paperpile.com/g/academic-search-engines/.

Mian, Imdad Ahmad et al. (2022). "A comprehensive skills analysis of novice software developers working in the professional software development industry". In: *Complexity* 2022.

Moores, Trevor T and Jerry Cha-Jan Chang (2009). "Self-efficacy, overconfidence, and the negative effect on subsequent performance: A field study". In: *Information & Management* 46.2, pp. 69–76.

Myers, Brad A and Jeffrey Stylos (2020). *Improving API Usability*. https://www.cs.cmu.edu/~NatProg/papers/p62-myers-CACM-API_Usability.pdf/.

Nielsen, Jakob (1994). *10 Usability Heuristics for User Interface Design*. https://www.nngroup.com/articles/ten-usability-heuristics/.

Olson, Gary M and Judith S Olson (2003). "Psychological aspects of the Human use of Computing". In: *Annu. Rev. Psychol* 54, pp. 491–516.

O'brien, Michael P (2003). "Software comprehension–a review & research direction". In: *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report*.

Pirolli, Peter and Stuart Card (1999). "Information foraging." In: *Psychological review* 106.4, p. 643.

Postman (2020). *2020 State of the API Report*. https://www.postman.com/state-of-api-2020/.

Qualtrics (2021). *What is customer effort score (CES) how do I measure it?* https://www.qualtrics.com/uk/experience-management/customer/customer-effort-score/?rid=ip&prevsite=en&newsite=uk&geo=NL&geomatch=uk.

Raman, Sowmyan (1998). "Lean software development: is it feasible?" In: *17th dasc. aiaa/ieee/sae. digital avionics systems conference. proceedings (cat. no. 98ch36267)*. Vol. 1. IEEE, pp. C13–1.

Rijke, Wouter J et al. (2018). "Computational thinking in primary school: An examination of abstraction and decomposition in different age groups". In: *Informatics in education* 17.1, pp. 77–92.

Robins, Anthony, Janet Rountree, and Nathan Rountree (2003). "Learning and teaching programming: A review and discussion". In: *Computer science education* 13.2, pp. 137–172.

Saenz, Juan Pablo and Luigi De Russis (2022). "On How Novices Approach Programming Exercises Before and During Coding". In: *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pp. 1–6.

Sánchez Carracedo, Fermín et al. (2018). "Competency maps: an effective model to integrate professional competencies across a STEM curriculum". In: *Journal of Science Education and Technology* 27.5, pp. 448–468.

Schefer-Wenzl, Sigrid and Igor Miladinovic (2019). "Developing Complex Problem-Solving Skills: An Engineering Perspective." In: *International Journal of Advanced Corporate Learning* 12.3.

– (2020). "Developing 21st century skills in engineering studies with E-learning". In: *Int. Conf. E-Learn. Workplace*. Vol. 2020, pp. 1–3.

Sedgwick, Philip and Nan Greenwood (2015). "Understanding the Hawthorne effect". In: *Bmj* 351.

Sedhain, Abim and Sandeep Kaur Kuttal (2022). "Information Seeking Behavior for Bugs on GitHub: An Information Foraging Perspective". In: *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 1–3.

Seel, Norbert M, Sabine Al-Diban, and Patrick Blumschein (2000). "Mental models & instructional planning". In: *Integrated and holistic perspectives on learning, instruction and technology: Understanding complexity*, pp. 129–158.

Storey, M-AD, F David Fracchia, and Hausi A Müller (1999). "Cognitive design elements to support the construction of a mental model during software exploration". In: *Journal of Systems and Software* 44.3, pp. 171–185.

Thangarajoo, Yagulawathi and A Smith (2015). "Lean thinking: An overview". In: *Industrial Engineering & Management* 4.2, pp. 2169–0316.

TomTom (2022). *TomTom | Home*. https://www.tomtom.com.

Twitter (2023a). *Get trends near a location*. https://developer.twitter.com/en/docs/twitter-api/v1/trends/trends-for-location/api-reference/get-trends-place.

– (2023b). *Search Tweets: Standard v1.1*. https://developer.twitter.com/en/docs/twitter-api/v1/tweets/search/api-reference/get-search-tweets.

UserTesting (2022). *Hear what your audience is saying and see what they mean*. https://www.usertesting.com/.

Vaccari, Lorenzino et al. (2021). "APIs for EU Governments: A Landscape Analysis on Policy Instruments, Standards, Strategies and Best Practices". In: *Data* 6.6, p. 59.

Van Deursen, Arie et al. (2003). "Experiences in teaching software evolution and program comprehension". In: *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, pp. 283–284.

Wang, Yingxu and Vincent Chiew (2010). "On the cognitive process of human problem solving". In: *Cognitive systems research* 11.1, pp. 81–92.

Appendix A: Example response GET Request


```
[
  {
    "trends": [
      {
        "name": "#GiftAGamer",
        "url": "http://twitter.com/search?q=%23GiftAGamer",
        "promoted_content": null,
        "query": "%23GiftAGamer",
        "tweet_volume": null
      },
      {
        "name": "#AskCuppyAnything",
        "url": "http://twitter.com/search?q=%23AskCuppyAnything",
        "promoted_content": null,
        "query": "%23AskCuppyAnything",
        "tweet_volume": 14504
      },
      {
        "name": "#givethanks",
        "url": "http://twitter.com/search?q=%23givethanks",
        "promoted_content": null,
        "query": "%23givethanks",
        "tweet_volume": null
      },
    ],
    "as_of": "2020-11-20T19:37:52Z",
    "created_at": "2020-11-19T14:15:43Z",
    "locations": [
      {
        "name": "Worldwide",
        "woeid": 1
      }
    ]
  }
]
```

§

Appendix B: Working example in Strategy B

CONFIDENTIAL: This content includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

Appendix C: Strategy B in full screen

CONFIDENTIAL: This content includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

Appendix D: Developer Journey

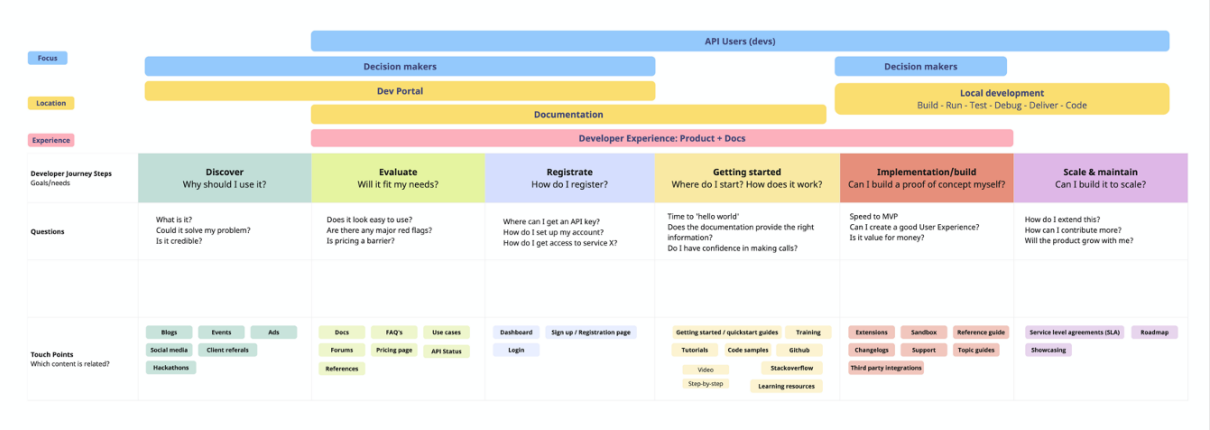

FIGURE D.1: Developer Journey

Appendix E: Consent Form

Informed Consent of Participation

You are invited to participate in the user study about solving complex software problems. Initiated and conducted by Thomas Nolst Trenité. The research is supervised by Prof. Thomas Kosch. Please note:

- Your participation is entirely voluntary and can be withdrawn at any time
- The user study will last approximately 60 minutes
- We will record personal demographics (age, degree, etc.)
- We will record the screen and activities on this device that are necessary for the study and take notes afterwards.
- All records and data will be subject to standard data use policies
- All records and subject-related data will be anonymized
- Repeated participation in the study is not permitted

The alternative to participation in this study is to choose not to participate. If you have any questions or complaints about the whole informed consent process of this research study or your rights as a human research subject, please contact Prof. Thomas Kosch (E-Mail: thomaskosch90@gmail.com). You should carefully read the information below. Please take the time you need to read the consent form.

1. Purpose of this Research

The purpose of this research is to measure the effectiveness of proposed support strategies in solving complex software problems. Your participation will help us achieve this goal. The study is not a test of your skills and focuses more on completing the purpose above. The results of this research may be presented at scientific or professional meetings or published in scientific proceedings and journals.

2. Participation and Compensation

Your participation in this user study is completely voluntary. You will be one of approximately 30 people being tested for this research. You will receive a 15,- bol.com voucher as compensation for your participation. You may withdraw and discontinue participation at any time without penalty or losing the compensation. If you decline to participate or withdraw from the user study, no one will be informed. You can deny answering questions if you feel uncomfortable in any way. The investigator may withdraw you from this research if continued participation will not meet the study goals or affect your well-being.

3. Procedure

After confirming the informed consent the procedure is as follows:

1. You will fill in a questionnaire on demographic data and your estimated programming experience

2. I will start recording the screen and you will be reading the instructions and looking at the content of the folder provided to you at your desktop.
3. You will be completing three tasks where in between you will need to fill in a questionnaire. For completing the tasks you will be getting an API key, instructions and template code.
4. If you are done with the tasks or manage to complete a part of them, I will stop the recording and you will need to fill in a final questionnaire on the experience you've had
5. The session is ended and you will receive compensation.

The complete procedure of this user study will last approximately 60 minutes.

4. Risks and Benefits

There are no risks associated with this user study. Discomforts or inconveniences will be minor and are not likely to happen. If any discomforts become a problem, you may discontinue your participation. In order to minimize any risk of infection, hygiene regulations of the University Utrecht apply and must be followed. Any violations of the hygiene regulations or house rules of this institution can mean immediate termination of the study. If you get injured as a direct result of participation in this research, please reach out to the principal investigator. Enrolled students are automatically insured against the consequences of accidents through statutory accident insurance and with private liability insurance in case of any damages. The confirmation of participation in this study can be obtained directly from the researchers.

5. Data Protection and Confidentiality

We are planning to publish our results from this and other sessions in scientific articles or other media. These publications will neither include your name nor cannot be associated with your identity. Any demographic information will be published anonymized and in aggregated form. Contact details (such as e-mails) can be used to track potential infection chains or to send you further details about the research. Your contact details will not be passed on to other third parties. Any data or information obtained in this user study will be treated confidentially, will be saved encrypted, and cannot be viewed by anyone outside this research project unless we have you sign a separate permission form allowing us to use them. All data you provide in this user study will be subject of the General Data Protection Regulation (GDPR) of the European Union (EU) and treated in compliance with the GDPR. Subsequent uses of records and data will be subject to standard data use policies, which protect the full anonymity of the participating individuals. Faculty and administrators from the campus will not have access to raw data or transcripts. This precaution will prevent your individual comments from having any negative repercussions. Access to the raw interview transcript and transcribed observation protocol will be limited to the authors of this research, academic colleagues, and researchers with whom he might collaborate as part of the research process. Any interview content or direct quotations from the interview that are made available through academic publications or other academic outlets will be anonymized so that you cannot be identified. During the study, we log experimental data, record the screen and/or activities on this device, and take notes at the end of the user study. Raw data and material will be retained securely and in compliance with the GDPR, for no longer than necessary or if you contact the researchers to destroy or delete them immediately. As with any publication or online-related activity, the risk of a breach of

Appendix F: Questionnaires

Questionnaire 0 | Introductive questions

Determines participants' self estimated API usage and keeps track of demographic data

1. What is your age?

   Markeer slechts één ovaal.

   - ◯ < 18
   - ◯ 18 - 25
   - ◯ 25 - 30
   - ◯ > 30

2. What is the highest degree or level of education you have completed?

   Markeer slechts één ovaal.

   - ◯ High School
   - ◯ Bachelor's Degree
   - ◯ Master's Degree
   - ◯ Ph.D. or higher
   - ◯ Prefer not to say
   - ◯ Anders: _____

FIGURE F.1: Questionnaire 0

3. What is your current employment status?

   Markeer slechts één ovaal.

   - ◯ Employed full-time (40+ hours a week)
   - ◯ Employed part-time (less than 40 hours a week)
   - ◯ Unemployed (currently looking for work)
   - ◯ Student
   - ◯ Retired
   - ◯ Self-employed

4. For how many years have you been programming?

   _____

5. On a scale from 1 to 5, how do you estimate your programming experience?

   Markeer slechts één ovaal.

	No experience					Highly experienced
1	◯					
2		◯				
3			◯			
4				◯		
5					◯	

FIGURE F.2: Questionnaire 0

Questionnaire 1 - Task 1

1. Overall, how difficult or easy was it to make the right API call using the TomTom documentation?

Markeer slechts één ovaal.

Very difficult

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

6 ◯

7 ◯

Very easy

FIGURE F.3: Questionnaire 1-3

2. Overall, how difficult or easy was it to build a working prototype using the data from the API?

Markeer slechts één ovaal.

Very difficult

1
2
3
4
5
6
7

Very easy

FIGURE F.4: Questionnaire 1-3

3. Rate your level of agreement to the following statement: *"The environment provided me with the right information to make the API call"*

Markeer slechts één ovaal.

Strongly disagree

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Strongly agree

FIGURE F.5: Questionnaire 1-3

4. Rate your level of agreement to the following statement: *"The environment helped me accomplish the programming task"*

Markeer slechts één ovaal.

	Strongly disagree
1	◯
2	◯
3	◯
4	◯
5	◯
	Strongly agree

FIGURE F.6: Questionnaire 1-3

Questionnaire 4 - Tasks finished

Some questions to round up the study, get general opinions and highlights

1. How confident do you feel in using the platform and tool(s) effectively?

Markeer slechts één ovaal.

Not at all confident

1

2

3

4

5

Very confident

FIGURE F.7: Questionnaire 4

CONFIDENTIAL: The remaining content from Questionnaire 4 includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

Appendix G: Bar Charts

CONFIDENTIAL: This content includes confidential information belonging to TomTom, and as such, it cannot be disclosed or shown to the public. This information is proprietary and protected under non-disclosure agreements.

Appendix H: Interview Scripts

Interview Script (acquainted)

Intro

Thanks for participating in this interview! Today I want to chat a bit about what work you do using the TomTom API in general and hear more about how you use the API. There are no right or wrong answers to any of my questions. This is meant to be a free conversation where you're the expert and I'm learning from you.

Before we start, do you mind if I record this session? The recording won't be shared externally – it's only for my internal notes.

Background Questions

1) What is your name and age?
2) How many years of (API) programming experience do you have?
3) Can you confirm that you are familiar with the TomTom API? As in: you are familiar with the overall architecture and some details of the system. You know our different API offerings and what they can do.
    a) How long did the process take for you to become familiar with the general architecture and details (like functions and data requests) of the system?
4) Outside of TomTom: What API's have you used before? Would you consider your API experience as Little, Average, Advanced, or Expert?

Questions related to the TomTom API

5) Can you please tell me a little bit about your role and what work you do regarding the TomTom API services?
    a) What API's do you use?
    b) How frequently do you use them?
6) Could you provide me with an example use case of where one of the API's would come into play in a personal project or application build.
7) How do you use the API design documentation to better comprehend how to build applications?
8) What parts of the API do you currently struggle with?

Information relevance

9) If you use our documentation to get the relevant information you need to build an application..What parts are relevant to you and which are not?
10) Do you have any expectations for certain elements (such as code fragments) to exist in this page?

Within my thesis I'm investigating where complexity plays a role in the current API services. We can describe complexity as how easy or difficult it is to use a particular API based on the public functions and data of this API.

Questions w/ regards to complexity

11) How do you come across system or interface complexity when working with the APIs?
    - Could you also provide a detailed example of when you experienced this?

FIGURE H.1: Script for interviewing API-familiar developers

- But you do have some experience with the API. Just out of curiosity: Do you have a method to still be able to deal with complexity when working with our API's and if so, what method(s) do you use?
- What tools would you mostly use when trying to integrate an API that can help you with making things easier to comprehend?

By improving the API design through support strategies, we will then try to simplify this complexity and make it more comprehensible.

12) Do you feel like there exists enough support for you to try and understand these functions and data requests better to help dealing with complexity?

Questions to round up

13) Is there anything you would like to add to what you've already said?
14) Do you perhaps have any contacts for me to reach out to that are familiar with API's but not with the TomTom API in specific?
15) In a few months, we will be testing the opposed supportive strategies with internal developers as well in a small pilot study. Would you be willing to participate?

Thank you so much for making the time to have this talk with me. Your answers were really valuable for this research. Would you add anything to the current existing questions? Maybe something you think would be really interesting to ask other developers?

FIGURE H.2: Script for interviewing API-familiar developers

Interview Script (NOT acquainted)
INTRODUCE SELF!
Intro
Thanks for participating in this interview! Today I want to chat a bit about what work you do using the TomTom API in general and hear more about how you use the API. There are no right or wrong answers to any of my questions. This is meant to be a free conversation where you're the expert and I'm learning from you.

Before we start, do you mind if I record this session? The recording won't be shared externally – it's only for my internal notes.

Background Questions

1) What is your name and age?
2) How many years of programming experience do you have?
3) What API's have you used before? What would you say your API experience is?
4) Can you confirm that you are **NOT** familiar with the TomTom API? (not familiar with the overall functions, data and architecture).

I would like to know more about how developers like you explore the offering of our API's. Therefore,

5) What API have you used recently? How do you normally navigate through this API structure and what information do you use in that process? What is your experience with this?

Discovery Browsing

6) Free exploring (15m): Could you navigate through the TomTom API offering for me and see if there is anything interesting for you to find? You don't have to look for something specific, just see what's out there. Also, if something interests you, feel free to stay there as long as you prefer. For me, it would help if you think out loud when you are going through this because what you say, do, think and feel is very important for our research.
    a) Take ubereats example
7) What is different in your information finding process compared to what you are used to? Are there similarities?
8) Did you experience comprehension of the TomTom API? Did you find it difficult or easy? Why?
    a) What can be improved?
    b) Do you have an example API you have used before that you found easy to comprehend?
    c) **Vague question:** How do you usually understand code you are not familiar with?

Recall

9) How likely is it that the pages you discovered will provide an answer to your questions?
    a) How long is it going to take to get the answer if you go to that page?

FIGURE H.3: Script for interviewing API-unacquainted developers

Questions to round up

10) Is there anything you would like to add to what you've already said?
11) Would you add anything to the current existing questions? Maybe something you think would be really interesting to ask other participants?

Thank you for making the time to have this session with me. Your answers were really valuable for this research.

FIGURE H.4: Script for interviewing API-unacquainted developers