Institute for Psychology
Faculty for Psychology und Sport Science
Leopold–Franzens–University Innsbruck

# *Aligning Mental Models and Models Used in Software Development*

Bachelor's Thesis
for obtaining the Academic Degree
Bachelor of Science (B. Sc.)
in the academic field Psychology

submitted by
*Bernhard Mayr, B. Sc., 11730144*

Supervisor: *Univ.-Prof. Dr. Pierre Sachse*
Innsbruck, *June 21, 2020*

# Acknowledgements

Even for a bachelor's thesis there is more to putting letters on paper. This would not be possible without people being supportive, inspiring and caring.

First of all I would like to thank my supervisor Mr. Pierre Sachse for the trust and straightforwardness in handling this bachelor's seminar in times of COVID-19. In a time like this I realized it was even more important that I could write about a topic I am passionate in. I am very grateful that I got the chance to write this thesis in the field of psychology of programming and it proved to me that this is the path that I want to take.

Sebastian, Andi, Bernhard, and Gregor, thank you for believing in me while creating the open source project Statecharts.NET at Innerspace GmbH over the last year, supporting my research in psychology with the technical implementation.

The biggest shout out at work goes to you Rebecca, for making me realize again how much I love teaching programming, pair programming and creating tools that make me love programming. Thanks for being my subject of investigation, it was a pleasure to work with you on Statecharts.NET.

No matter if in German or in English, thank you dad for all the awesome discussions, your historical knowledge, your patience and your proof-reading skills!

My friends know how much I love talking about nerdy stuff, thank you for talking about nerdy stuff as well. So many great ideas originated from these conversations! Without you Daniel, Max, Marlene, Iris, David, Christina, and Markus, this thesis would miss all the knowledge I gained by talking to you about Statecharts.NET.

Teresa, thanks for listening to my daily dose of statecharts, reminding me when it is enough, being there for me and being you!

B. M.

# Abstract

Despite of being a human activity, developing software usually does not receive the amount of psychological research needed. A major difficulty is the huge gap between these two disciplines being the reason for this missing interdisciplinary thinking. Developing software is way too often reduced to the act of entering commands in a way the computer does what the programmer wants it to do. In reality a lot of steps happen until software developers can express their ideas in such a clarity that they are able to create an executable and correct program. Looking at the software development process holistically and applying ideas of cognitive psychology, especially the ones of mental models, the author of this thesis tries to align the thought process behind creating software closer with the actual technical process of doing it. This leads to the research question: *Which findings of cognitive and social psychology support the usage of statecharts combined with hole-driven development for enabling better communication in software development processes, or how can these concepts be adapted to closer align to the way the human brain works?*

Keywords: mental models, psychology of programming, statecharts

# Kurzfassung

Die Entwicklung von Software ist eine äußerst menschenzentrierte Angelegenheit. Die Aufmerksamkeit, die ihr die Psychologie dabei schenkt, ist im Gegensatz verschwindend gering, was durchaus am sehr großen Unterschied dieser zwei Wissenschaftsbereiche liegt. Viel zu oft wird Programmieren auf das reine Erstellen von Quelltexten reduziert oder umgangssprachlich gar als Tippen von Nullen und Einsen bezeichnet. In Wirklichkeit sind die Prozesse, die zum Erstellen von Software notwendig sind, allerdings wesentlich komplexer und interdisziplinärer. Das Ziel dieser Arbeit ist die Betrachtung des Softwareentwicklungsprozesses aus der psychologischen Perspektive, wobei der Fokus auf der Erstellung und Weitergabe von mentalen Modellen und gemeinsamem Verständnis liegt. Die Forschungsfrage lautet daher: *Durch die Kombination von Statecharts und Hole-Driven-Development entsteht eine neue Art und Weise Software zu entwickeln. Welche Erkenntnisse der Kognitions- und Sozialpsychologie unterstützen diese Kombination bzw. wie kann die Softwareentwicklung so adaptiert werden, dass sie den Prozessen unseres Gehirns ähnelt?*

Keywords: Mentale Modelle, Psychologie der Computerprogrammierung, Statecharts

# Contents

# 1 Introduction

"A programmer does not primarily write code; rather, he primarily writes to another programmer about his program solution."

*Unknown*

Unfortunately the author of this rather social definition of a programmer's tasks was not handed down, nevertheless this article laid the foundation for human-centered programming (Anonymous, 1967). Reading Peter Naur's influential article "Programming as Theory Building" written in 1985 (Naur, 1985) this human-centered approach to programming seems to have been forgotten. Only 18 years after "What A Programmer Does" he states that: "[...] much current discussion of programming seems to assume that programming is similar to industrial production, the programmer being regarded as a component of that production, a component that has to be controlled by rules of procedure and which can be replaced easily" (Naur, 1985). At the same time the interdisciplinary research field *Psychology of Programming* originated (Myers & Ko, 2009), indicating that this mismatch of what programming is and as what it is seen was recognized.

## 1.1 Problem Definition

As stated in Anonymous (1967), programming is a communication process. Even if programming is not seen as a mainly human-centered activity, translating thoughts into a language understood by machines is a form of communication, namely *Human Computer Interaction* (HCI) (Myers & Ko, 2009). As software got more and more complex, the sole activity of programming transitioned to the much broader task of software development. This process begins with the task of requirements analysis. After acquiring a common understanding in the whole team, these requirements have to be implemented, tested and shipped to the customer (Mayr, 2005). This extremely simplified description of software development alone exhibits the difference to programming. Curtis and Walz stated: "The fact that this field is usually referred to as the ‘psychology of programming’ rather than the ’psychology of software development’ reflects its primary orientation to the coding phenomena.” (Curtis & Walz, 1990, p. 267) which shows that this mismatch between programming and software development is not unknown to the psychology of programming.

Modern software development processes underlie an iterative and incremental nature (Mayr, 2005) which means that user feedback is constantly incorporated into the development cycle. New requirements arise or existing ones are refined, in either case the software under development has to be adapted. The basis for this adaption is communication, a human factor, that does not come without its difficulties (see Section 2.1). As stated in Curtis and Walz (1990) mainly organizational processes characterize software development.

*Source of truth* is a saying commonly used in software development for the place where something is defined. Good software design favors a *single* source of truth, because then changing the behavior means changing it only in one place. Applying this idea to the final product of the software development process the single source of truth is the actual source code. No matter what a requirements document defines or what the user documentation states, if these are not in sync with the source code, they “lose”, the code *speaks the real truth.*

So why not adopt Schraube’s ideas (Schraube, 2012) that technology influences people the same way they influence technology and apply it in a way that establishes a technology-based rethinking process inducing a change in organizational processes.

## 1.2 Research Question

The problem as previously described is usually not perceived explicitly in the software development industry, it’s just the characteristics of established software development processes and inherently caused by the way software is currently created. In Bret Victor’s words this *problem* might rather be stated as a *principle* that software development does not have to be like that (Victor, 2012). The upcoming ideas emerged from the author’s psychology internship at the company Innerspace GmbH in Wattens, Austria[1]. Generally employed as a software architect the author switched perspectives for two months and studied the software development process from a more psychological view. The company Innerspace creates virtual reality trainings for pharmaceutical companies. One main difference to traditional web or desktop application development is the diversity of the backgrounds of the people involved in the process of creating Virtual Reality trainings.

[1]https://www.innerspace.eu/

In addition to software developers, 3D artists, training designers, narrators and game engineers are needed for content creation. As already stated in Section 1.1, at the end the source code is the actual source of truth. Creating a consistent product of high quality, the 3D assets, narrations and training flows have to be aligned closely to the program's source code as they are also artifacts of the development process.

While researching possible technical solutions for the problem of implementing the application flow of complex applications like Virtual Reality trainings, the author stumbled upon the rather old concept of *statecharts* (Harel, 1987). Statecharts are a "visual formalism for complex systems" (Harel, 1987). In more general words this means, that statecharts describe a mathematically proven visual language for creating computer programs. The notation is based on a combination of flowcharts and state diagrams and enables a visual representation of logic in computer programs. Furthermore the visual language of statecharts provides simulation capabilities and therefore enables people to visually see and experiment with the logic of the program, without actually using the program. In fact, the program might not even exist at this point, but people can collaboratively design what the program should do. The main hypothesis is that the nature of an executable visual model allows the participants in a software project to create a common mental model, resulting in a new way of communication that prevents misunderstandings early on.

To further improve this communication process, the fractal nature of statecharts (logic can be refined from abstract to concrete) is the perfect fit for hole-driven development (see Section 2.3). This relatively new idea of writing computer programs can be compared with a "fill in the blank"exercise. Instead of having to write a fully completed program, developers can leave holes for the computer and then ask it to provide help based on mathematical reasoning. Applying this idea at the level of human interaction, the second hypothesis is that introducing holes in statecharts (enabled by their fractal and composable nature) enables a new way of communicating, defining and refining requirements.

This leads to the research question: *Which findings of cognitive and social psychology support the usage of statecharts combined with hole-driven development for enabling better communication in software development processes, or how can these concepts be adapted to closer align to the way the human brain works?*

After introducing the technological concepts in Chapter 2 and the psychological foundations in Chapter 3 these findings will be combined and presented in Chapter 4 followed by a concluding discussion in Chapter 5.

# 2 Technological Foundation

"I became convinced from the start that the notion of a state and a transition to a new state was fundamental to their thinking about the system."

*David Harel*

Crossing the interdisciplinary boundaries between programming and psychology needs the definition of the technical concepts that are referenced. For example knowing what is meant by *state* and *transition* in David Harel's above-mentioned psychological discovery is necessary to understand why this might impact the way people think about systems. As stated by Schraube (2012), the interaction between a person's psyche and technology is always bidirectional, we get shaped by technology the same way we shape it.

## 2.1 Characteristics of Software Development Processes

Upon answering the question on how communication in software development processes can be improved using technology, it is important to define these processes, take a quick look at their history and examine the difficulties of changing requirements.

The complex process of software development involves a lot of tasks, stakeholders and uncertainties (Mayr, 2005). Being able to plan, organize and control this process, different paradigms of organizing the process of software development were established and improved. Independent of the software development paradigm, six main phases can be identified, namely: *requirements analysis*, *specification*, *design*, *implementation*, *testing* and *maintenance* (Harel, 1987). Somehow these phases have to be completed in that order, albeit the paradigms differ in the way the phases are went through, The most notable differences are the flexibility in adapting to changing requirements, customer communication, the way phase transitions are handled, how fast the project iterates and how often intermediate results are delivered (Mayr, 2005).

### 2.1.1 Traditional Paradigms vs. Agile Paradigms

As described in Mayr (2005), the evolution of software development paradigms underwent a lot of change. Over time the characteristics changed from rigorous (traditional, heavy and stable) to agile (modern, lean, fragile), with the agile movement based on Fowler and Highsmith (2000) as the current status quo (Ousterhout, 2018). In comparison to the documentation-heavy traditional paradigms, agile paradigms got rid of documentation overhead, heavily rely on prototyping and follow an iterative-incremental model (Mayr, 2005). This iterative-incremental model is characterized by short consecutive development cycles and a high frequency in delivering intermediate products, thus staying in close contact with the customer, making sure that the right product is delivered and requirements are understood correctly (Mayr, 2005).

Dispensing long requirements analysis and specification phases in favor of a short time to market affects software architecture and design, subsequently complicating the creation of well-architectured, solid software. Ousterhout (2018) introduces the terms *tactical programming* and *strategic programming.* While the main focus of tactical programming is to create something that works, the focus of strategic programming is to create something that lasts. In the end this is a business decision, but the effects it has on software development have a huge impact regarding the design of stable, well-constructed systems (Ousterhout, 2018).

### 2.1.2 Requirements Analysis and Changing Requirements

Requirements are properties that have to be fulfilled by a system (Mayr, 2005). Requirements analysis is the process of understanding what the customer needs and translating these to a specification upon which the software program can be built. It is necessary that these identified requirements can be validated and verified.

As stated in Mayr (2005) the difficulties of requirements analysis are often underestimated. Two of these are of high importance for this thesis. Requirements analysis is a human-centered communication process. Analysts have to understand the customer's domain, excel at communication with various stakeholders and create a form of specification that is understood by all participants as well as traceable from the origin of a requirement into the finished product. As Curtis and Walz (1990, p. 265) discovered: “That is, many projects spen[d] tremendous time rediscovering information that, in many cases, had already been generated by customers, but not transmitted.” Changing requirements caused by the customer, by competitors or by new market innovations are another big problem for requirements analysis. As stated by Mayr (2005), requirements analysts' methods and notations should favor an easy way of adapting new and changing requirements.

Even if the analysts' methods can adapt to changing requirements, the difficulty of communication still exists and gets intensified by rapidly changing requirements. More closely aligning the methods and notations of analysts to those used in the implementation phase can reduce these costly communication problems. According to Peter Naur the designer's (analyst's in this case) job is not to pass along "the design" but to pass on "the theories" driving the design (Naur, 1985) preventing documentation being an "auxiliary, secondary product."

Leveson, Heimdahl, Hildreth, Reese, and Ortega (1991) report an interesting experience on requirements specification while working on an aircraft collision avoidance system called *TCAS II*. After realizing that the *minimal operational standards* document (MOPS) initially created specified the behavior too loosely, an industry/government alliance started working on a more detailed specification in English. At the same time Leveson et al. (1991) started working on a formal specification. After only one year the English specification was dropped in favor of the formal specification, because of not being able to handle the complexity in a correct way. Admittedly adopting the formal specification brought its problems as well. There is a sweet spot in requirements specification described by Leveson et al. (1991): "The specification must be formal enough [...] to use as a basis for a safety analysis. It must also be readable enough for noncomputer experts to read and review and be usable for both building and certifying [...] systems."

### 2.1.3 Insights from Psychology

Kitchenham and Carn identified the uniqueness of the software development process in the following way: "Closer inspection of the software production process suggests that it is an engineering discipline like any other engineering discipline. It is not as mature as electrical or chemical engineering or even agriculture. Nonetheless, it is *not* an art form. Before software engineering can mature as an engineering discipline, practitioners need a better understanding of the process by which software is created in response to a demand and of the risks and errors which are associated with the process." (Kitchenham & Carn, 1990, p. 274), conducting that the major difference to other engineering disciplines is the missing maturity and empirical knowledge.

While most software development process paradigms originate in economics, the layered behavioral model (Fig. 2.1) of software development (Curtis & Walz, 1990) emphasizes the cognitive, social and organizational processes. They stated that "the layered behavioral model focuses on the behavior of the humans creating the artifact, rather than on the

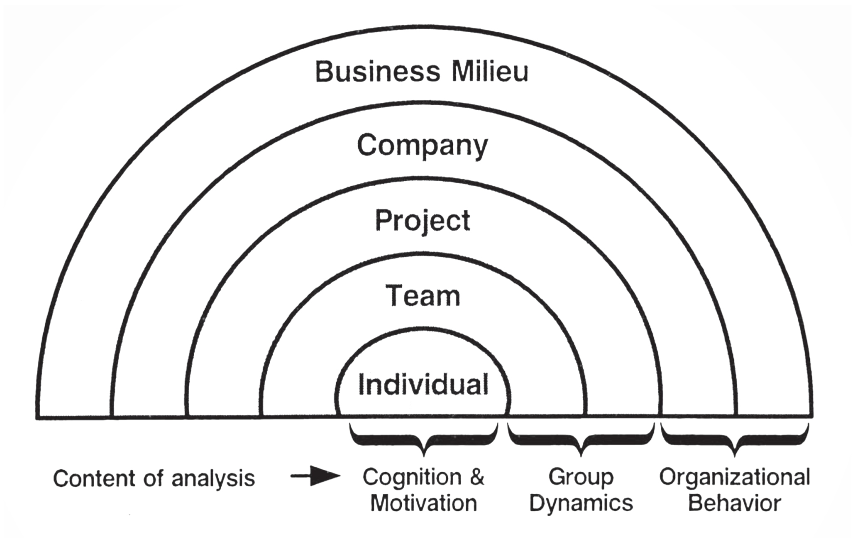


Figure 2.1: Layered Behavioral Model of Software Development

evolutionary behavior of the artifact through its developmental stages" (Curtis & Walz, 1990, p. 254). Confirming Curtis' findings, Gerald M. Weinberg already presented empirical studies about "Programming as a Social Activity" in 1971 (Weinberg, 1971). His studies also focus on the much understated social aspects of programming while most of the focus is put on tools for helping the individual developer, and forgetting about group, team and project communication.

Regarding communication, Curtis and Walz (1990) introduce the concept of a *boundary spanner*. During their empirical studies about the social aspects of psychology of programming they identified: "Boundary spanners translated information from a form used by one team into a form that could be used by other teams" (Curtis & Walz, 1990, p. 264). They become "hubs for information" (Curtis & Walz, 1990, p. 264) introducing the problem of implicit hidden knowledge that is not documented. Regarding the title of this thesis, boundary spanners "align mental models and models used in software development", they transmit and translate knowledge. The goal is not to replace boundary spanners, but merely inviting more and more people in the software development process to take part in transferring knowledge, spanning boundaries and talking about the same things.

## 2.2 Statecharts

Invented in 1983 by David Harel, statecharts are "visual formalism for complex systems" (Harel, 1987). While working on avionic systems for the Israeli Air Force, David Harel was asked to help taming the complexity of reactive systems. A reactive system is dominated by its “[...] reactivity; its event-driven, control-driven, event-response nature, often including strict time constraints, and often exhibiting a great deal of parallelism. A typical reactive system is not particularly data intensive or calculation intensive” (Harel, 2007). Reading this pretty narrow technical definition of a reactive system one might question the relevance of statecharts in everyday software. Ian Horrocks provides an answer to this question by specifying the *event-action paradigm* (Horrocks, 1999). User Interfaces are event-driven, they present screens with data and wait for user interactions. Based on these they react, thus fulfilling all the requirements for a reactive system. And because every software that is interacted with has some kind of user interface, all these programs can utilize statecharts as a way of handling state.

Describing a visual notation with words is contradictory, so let’s take a look at Fig. 2.2 taken out of the original paper introducing statecharts (Harel, 1987). This statechart describes part of the behavior of a digital wristwatch. The rounded rectangles describe states the system can be in and the labeled arrows between these describe the state transitions. Basically a statechart is an enhanced mixture between state diagrams and flow charts. This also explains where the name comes from, according to David Harel the term *statecharts* was the only unused combination of “’state’ or ’flow’ with ’chart’ and ’diagram”’ in 1983.

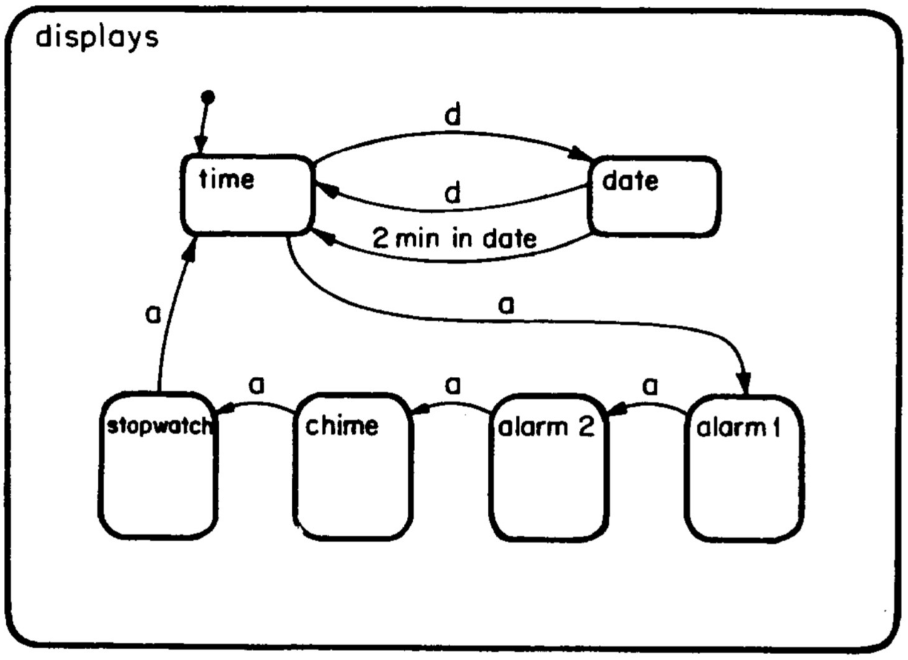


Figure 2.2: Example Statechart of a Digital Wristwatch Interface

As T.G.R. Green defines in Green (1982, p. 3) "A *program* is a succinct description of a temporal process." Martin Glinz describes why statecharts perfectly fit Green's definition of a program: "Naturally, statechart models will concentrate on requirements concerning dynamic system behavior and interaction" (Glinz, 2002, p. 1). Taking a look at Fig. 2.2 this statechart transmits this information in the following way:

- The outermost box *displays* is a compound state containing multiple sub-states such as *time*, *data*, *alarm 1* and so on. In statecharts a compound state is defined by having multiple sub-states with exactly one being active at a certain point in time.
- The arrow originating in a black dot leading into *time* specifies that this state is the initial active state of *displays*.
- The labels *d* and *a* name events that can occur in the system. In this particular example by David Harel the digital wrist watch contains four buttons, that upon pressing raise the events *a*, *b*, *c* and *d*.
- The arrows labeled *d* between *time* and *date* can be translated to a toggle functionality between the *time* and *date* states of the watch.
- The arrow labeled with *2 min in date* follows exactly its description. If the state *date* has been active for two minutes and no event occurred, the system automatically switches to the state *time*.
- If the system is in state *time* and the event *a* occurs it will transition to *alarm 1*, on another press of *a* into state *alarm 2* and so on.

One of the most important properties of statecharts in comparison to textual source code is the obviousness of valid and invalid transitions. Users of the statechart can immediately conclude which events are possible in which state. Another property of statecharts is that events not allowed in the currently active state are simply ignored instead of causing errors, thus leading to more robust and less faulty programs (Horrocks, 1999). For a full overview of the available features of statecharts please refer to Harel (1987).

### 2.2.1 The Goals Behind

In 2007, David Harel published his personal story behind statecharts titled "Statecharts in the Making: A Personal Account" (Harel, 2007). In this paper he talks about his psychological ideas and observations during the development of statecharts in the eighties.

Apart from creating a *visual formalism* for reactive systems, he wanted to create a notation for *model-based development* (which became really popular with the Unified Modeling Language [Cook et al. (2017)]) and an executable model where the behavior "can and should be executed just like [in] conventional computer programs [...]" (Harel, 2007, p. 1). This technical goals lay the foundation for two of his greater goals that can be attributed to social psychology (Harel, 2007, pp. 3–4):

- "The goal was to try to find, or to invent for these experts, a means for simply saying what they seemed to have had in their minds anyway."
- "The goal was to find a way to help take the information that was present collectively in their heads and put it on paper, so to speak, in a fashion that was both well organized and accurate."

The above mentioned *executability* is another fundamental idea of statecharts (Harel & Politi, 1998, p. 7), because of the simulation aspect taking a lot of cognitive load off the developer. Back in the eighties executable models weren't the standard, because the graphics and input capabilities of computers were not comparable to the ones used today, but even then David Harel imagined "[...] graphical workstations with large (blackboard size?) displays of fantastic resolution [...]" (Harel, 1987, p. 272). More than 30 years later, devices like the Microsoft Surface Hub[1] exist that perfectly match this description. Combined with Bret Victor's ideas of instant code executability (Victor, 2012) this could be the perfect match for an interactive development environment based on statecharts.

### 2.2.2 Aligning Thoughts with Fractal Architectures

Comparing different approaches to user interface development, André Staltz coined the term *fractal architecture* as "[an] architecture is said to be fractal if subcomponents are structured in the same way as the whole is" (Staltz, 2015). Fractals are a mathematical concept of geometric figures that appear the same at different levels.

As described earlier, statecharts feature the concept of compound states, wherein states can be nested inside each other. This property perfectly maps to "geometric figures that appear the same at different levels" and André Staltz's definition of fractal architectures. This state nesting allows to refine the behavior of a certain state by adding sub-states and thus adapting to new requirements without breaking others (Harel, 1987). Leveson et al. (1991) discuss the two main strategies of transforming requirements into software: *top-down*

[1]https://www.microsoft.com/en-us/surface/business/surface-hub-2

and *bottom-up*. The top-down strategy is based on top-down refinement where requirements are deconstructed one by one, starting from the most abstract one, transitioning down the tree of requirements. Unfortunately, this process only works for experts possessing a huge knowledge in software design and the problem domain. The bottom-up strategy, on the other hand, is characterized by less upfront thoughts about design and immediately starting with the implementation of requirements at a very detailed level. A major problem using this approach is that a well designed software architecture is nearly impossible to reach, especially as applications get larger (Horrocks, 1999). As stated in Leveson et al. (1991), programmers constantly switch between those two strategies while developing software.

This is exactly the point where the fractal nature of statecharts can be aligned with the characteristics of programming. States itself can be nested (*bottom-up*) as well as refined using sub-states (*top-down*), directly resembling the composition and decomposition of requirements as shown by Leveson et al. (1991). Glinz (2002) already started research in the direction of transforming requirements into statecharts, but only from a technical and not from a psychological perspective.

## 2.3 Hole-Driven Development

The term *Hole-Driven Development* is not yet scientifically defined, but based on other programming concepts like *Type-Driven Development* (Brady, 2017) or *Test-Driven Development* (McCracken, 1957). The main idea stems from the *Agda* programming language[2] where *holes* are called *goals*. Apart from Agda, other languages such as Haskell[3] and Idris[4] feature the concept of holes. The common properties of those languages is their functional nature and a very sophisticated type system. These two aspects lay the foundation for Hole-Driven Development as currently understood.

Simply said, a *hole* declares a missing part in an application. Gamari (2019) contains the up to now most comprehensive description of the concept of holes, while Brady (2017) created the example shown in the following section.

[2]https://wiki.portal.chalmers.se/agda/pmwiki.php
[3]https://www.haskell.org/
[4]https://www.idris-lang.org/

### 2.3.1 Example in Idris

The Idris program in Fig. 2.3 demonstrates the simplest possible hole. In Line 2 the function `main` is defined that should print something (using the standard library function `putStrLn`) to the console. The "hole" part of this program is `?greeting`. Signaling a hole, the question mark tells the Idris compiler that something in this program is missing that the developer didn't specify yet. The syntax using a question mark resembles a question to the compiler.

```
1 main : IO ()
2 main = putStrLn ?greeting
```

Figure 2.3: Hole-Driven Development in Idris

Edwin Brady coined the phrase "the compiler as your lab assistant" (Brady, 2017). His imagination of a compiler is one of a counterpart one interacts or pair-programs with. One tells the compiler all the things that are currently in ones mind and as a reward one can ask questions and the compiler will try to answer those based on the knowledge already provided. In terms of psychology, this is a pretty human way of interacting with the computer.

A practical example of this ideas looks like the following output of Idris' interactive command line interface. Using `:t greeting` one can ask the compiler for the type of the hole "greeting". As an answer one gets `greeting : String` which signals that the missing value is of type String, or in a non-technical terminus simply a text to complete the program definition.

```
*Hello> :t greeting
--------------------------------------
greeting : String
```

In contrast to the previous example, if you just try to use `greeting`, the Idris compiler tells you that you have a hole in your program labeled *greeting* of type String. This shows the communication process Edwin Brady talks about.

```
*Hello> greeting
?greeting : String
```

"Holes allow you to develop programs *incrementally*, writing the parts you know and asking the machine to help you [...]" (Brady, 2017, p. 21). Stepping a little bit back and comparing this way of developing software to how software development processes evolved (see Section 2.1.1), the similarities are unmistakable. Let's take a look at how holes are simulated in programming languages that don't support this feature natively before we get to psychological observations that were already discovered in the field of psychology of programming.

### 2.3.2 Simulating Holes

In languages that do not provide the feature of typed holes, programmers tend to come up with alternative solutions to holes without knowing of their existence. Psychological research (see Section 2.3.3) indicates that the process of creating these holes is so natural to the thought process that even within languages without the concept of holes, developers find a way of expressing them. There are mainly two ways of simulating holes:

1. One might throw exceptions such as `NotImplementedException`, as documented in Microsoft (2020), to signal that a feature is not yet implemented.
2. Comments are another common way of deferring development of a particular method or feature, marking the comment as a "hole" prefixing it with `TODO:`. E. g. in C#[5] this looks like `// TODO: Load state from previously suspended application`. Interactive development environments are able to parse this information and provide an overview of to-dos as shown in Hogensen and Jacobs (2019).

These two concepts might look like sufficient surrogates at first, so what are the differences to real holes?

- The previously described interaction with the compiler is missing completely.
- By using exceptions the program crashes at run-time, which results in buggy software. Additionally, there is no real overview of all the simulated holes. One has to do a text-based search for the exception name.
- Utilizing comments doesn't result in the program crashing, rather not adhering to the specifications without any note at run-time. This might lead to incorrectly behaving software.
- Both approaches act like a "do-it-later" approach, without actively reminding the developer at a later time.

[5]https://docs.microsoft.com/en-us/dotnet/csharp/

### 2.3.3 Incomplete Thoughts

In the meta-analysis “Expert Software Design Strategies” conducted by Visser and Hoc in 1990, they collected thinking processes and strategies of expert programmers (Visser & Hoc, 1990). Comparing breadth-first and depth-first design methodologies, they found out that expert developers vary in their usage and most of the time combine both strategies. One commonality was the observation that the developers under research were observed making “notes to themselves” (Visser & Hoc, 1990, p. 241). Experts usually excel at maintaining relevant information in the working memory, nevertheless this memory capacity is limited and working on one problem might relate to other problems until there is not enough capacity left and they start taking those notes. They conclude that “[...] before introducing the constraints of actual programming languages, experts very often use[d] a personal pseudo-code” (Visser & Hoc, 1990, p. 242).

Combining this research, it seems plausible that real holes enhanced with comments could lead to a programming language that lets developers specify the already thought-out ideas in executable source code while still letting them express their ideas in written natural language with the advantage of the programming environment actually caring about their ideas.

# 3 Psychological Foundation

"Sometimes our tools do what we tell them to do. Other times, we adapt ourselves to our tools' requirements."

*Nicholas Carr*

Being able to create and shape the tools, why adapt to their requirements? Combining psychology with any other academic field enables an enhancement of understanding and thus the invention of tools that align closer to how humans think and behave.

## 3.1 Psychology of Programming

Psychology and programming are often not linked together by "of", rather by "in". The resulting academic fields include Human Computer Interaction (HCI) or User Experience (UX). Instead of focusing on the interaction between humans and computers, the *psychology of programming* cares about the cognitive and social aspects of programming itself. The target of this thesis (and the underlying research) is not the end-product of programming, but the act of programming.

### 3.1.1 History

Established in the late 1970s and early 1980s, researchers realized that programming tools and technologies should be evaluated based upon their ease of usage. The human aspect of tools as well as their cognitive effects became more and more important as computational power increased (Sajaniemi, 2008). As Curtis and Walz stated about the state of integrating social, organizational, ecological and interactional psychology into programming in 1990: "Disappointingly little theory and few research paradigms from these fields have been imported into the psychological study of programming" (Curtis & Walz, 1990, p. 253). Unfortunately, academic literature on the topic of psychology of programming did not really advance with only two relevant books available, titled *The Psychology of Computer*

*Programming* (Weinberg, 1971) and *Psychology of Programming* (Hoc, 1990). Despite this meager amount of literature in 1987 the Psychology of Programming Interest Group[1] (PPIG) was founded, still organizing annual workshops and bringing people together from diverse communities to discuss the importance of psychology of programming. As Sajaniemi (2008) describes, there is a dual character according the research motivation in psychology of programming. One the one hand, computer scientists are motivated to create new tools and improve existing ones based on knowledge gained from studying the psychological aspects of programming. On the other hand, psychologists are interested in new cognitive insights gained while studying programmers that can be applied to other domains.

### 3.1.2 Programming as Theory Building

In 1985, Peter Naur released his essay "Programming as Theory Building" that changed the way how programming was perceived (Naur, 1985). While the traditional (still broadly applied) perception of programming is the "production of a program and certain other texts", Naur's views of programming is that it "[...] should be regarded as an activity by which the programmers form or achieve a certain kind of insight, a theory [...]". As summarized and put into other words by Aranda (2011), programmers have to develop a shared understanding of the problem to solve. The actual program becomes a by-product of this process, thus the actual creation of source code losing importance. Supporting this view, Naur provides two examples of software being handed over to another team. If the main process of creating software is the production of source code and documentation, the project hand-over between two teams should be fairly easy. After an introductory phase, the new developers should be able to continue working on the product, creating new features, improving existing ones and fixing bugs. But as pointed out in Naur (1985, pp. 228–229), in both investigated cases the quality and performance decreased constantly after the hand-over, indicating that there must have been something lost during the transition from one team to the other. As Peter Naur put it into words: "Thus, again, the program text and its documentation has proved insufficient as a carrier of some of the most important design ideas" (Naur, 1985, p. 229).

The definition of theory used in this thesis is closely aligned to the one defined by Ryle (1984). *Theory* is understood as the knowledge a person possesses that does not only allow one to accomplish certain things, but also explain them, answer questions and argue about (Naur, 1985). This leads to the following observations (Naur, 1985, p. 234):

[1]http://www.ppig.org/

- "The death of a program happens when the programming team possessing its theory is dissolved."
- "Revival of a program is the rebuilding of its theory by a new programmer team."

Regarding changing requirements (see Section 2.1.2), Peter Naur's views differentiate strongly from those presented in Section 2.1.1, thus presenting a possible explanation for the difficulties of changing requirements. Keeping the option to modify programs late in the development process (develop for potentially changing requirements) is a synonym for program flexibility. Comparing flexibility to other engineering disciplines like the construction of buildings, modifications are extremely expensive and re-construction is often found to be preferable (Naur, 1985). Looking at programs through the perspective of programming as theory building, this analogy can be applied as well. Just because a program seems to consist of text files, adapting it cannot be accomplished by simply editing those files. Instead the underlying theory has to be modified and shared inside the programming team.

## 3.2 Mental Models

There are many possible ways of defining mental models (Herczeg, 2018). One possible definition that aligns well with the discipline of engineering is given by Rouse and Morris (1986, p. 7): "Mental models are mechanisms whereby humans are able to generate descriptions of system purpose and form, explanations of system functioning and observed system states, and predictions of future system states." Complementing this definition, mental models are based on beliefs, not facts, let people anticipate behavior of systems and are in flux (Dutke, 1994). Due to mental models being based on beliefs and the fact that a model abstracts reality, it is important to note that mental models might be erroneous and incomplete (Herczeg, 2018). However, mental models being in a constant state of flux indicates that they are flexible, temporary and constantly adapt to new information.

Mental models are heavily based on analogies (Dutke, 1994), where an analogy is a similarity in some respect between things that are otherwise dissimilar. Analogies can be conveyed using metaphors, didactic tools for pointing out analogies, often times utilizing visual similarities.

Another property of mental models as described in Dutke (1994) is their capability of simulation. This is possible, because mental models are procedural and not static. A process called *envisioning* creates a causal model which can then be "run" in the mind to simulate processes and predict future system states.

Due to the importance of mental models for the understanding of systems, mental model mismatches should be acted upon as soon as possible. Nielsen (2010) proposes two different ways of tackling mental model mismatches: “make the system compliant”, and “improve the users’ mental models”. Kitchenham and Carn (1990) state the problem of sharing ideas in programming as “[u]nderstanding a design developed by another designer often involves understanding a solution that you yourself would never have envisaged, given the particular problem statement.” Based on Peter Naur’s observations in “Programming as Theory Building” the other’s design or solution equals a theory, thus representing the other’s mental model. Sharing a common mental model developers “add code in ways that fit together” (Naur, 1985, p. 239). Developing a shared understanding needs the right documentation, according to Peter Naur, documenting what’s necessary to reconstruct the theory upon which the program was built. Based on the observations conducted by Naur (1985, p. 240) those are “metaphors”, “text describing the purpose of the major components” as well as “diagrams representing the interactions between them”. These findings will lay the basis for the implications presented in Chapter 4.

Reading David Harel’s essay “Statecharts in the Making” with this knowledge about mental models in mind, the main problem in the mentioned project at IAI (Israel Aerospace Industries) prior to introducing statecharts gets pretty obvious. All the development teams possessed different mental models that were not aligned to each other or, as Harel discovered after talking to the different teams (radar people, flight control people, communications people, ...), “each group had had their own idiosyncratic way of thinking about the system, their own way of talking, their own diagrams, and their own emphases”, according to Harel (2007).

# 4 Implications

"More effective tools could be built if we understood the creation process and knew what tools were needed to support human weaknesses and magnify human strengths."

*Barbara Kitchenham*

## 4.1 Towards Mental Models for Software Development

This quote by Barbara Kitchenham applies to tools in general. Speaking of tools used in software development Kitchenham and Carn (1990) further specify these as "thinking tools for thinking." In this chapter various ideas and findings are presented that build upon the foundation presented in Chapters 2 and 3.

As shown in Section 2.1, software development processes involve a lot of participants. The customer or user imagines the perfect product, the requirements engineer thinks in (user) stories, testers use acceptance criteria and developers think in source code. Each of them creates an own mental model trying to solve one part of the common problem. Approaching a common problem with different mental representations results in the use of different tools, thus speaking "different languages" requiring translations or cross-domain knowledge and understanding of the people involved.

The title of this thesis indicates the necessity of the alignment of mental models to models used in software development. Considering specific mental models or models used in software development and creating counterparts would solve these problems in specific cases. Of much greater impact would be the creation of a software modeling tool that is aligned to the nature of mental models, thus enabling modeling in software development in such a way that it is analogous to developing mental models.

## 4.2 Combining Statecharts and Hole-Driven Development

To the knowledge of the author, statecharts and hole-driven-development have not yet been combined. By integrating the concept of holes into the notation of statecharts, the visual formalism as introduced by Harel (1987) is extended by the property of *incompleteness*. Fig. 4.1 exemplifies how such an extension may look like. The goal of this thesis is not to invent a notation for such an extension. Merely the author explains ways of how such a combination could improve the process of software development backed by psychological research. In comparison to Fig. 2.2, the state *date* is now replaced by a *state hole* named

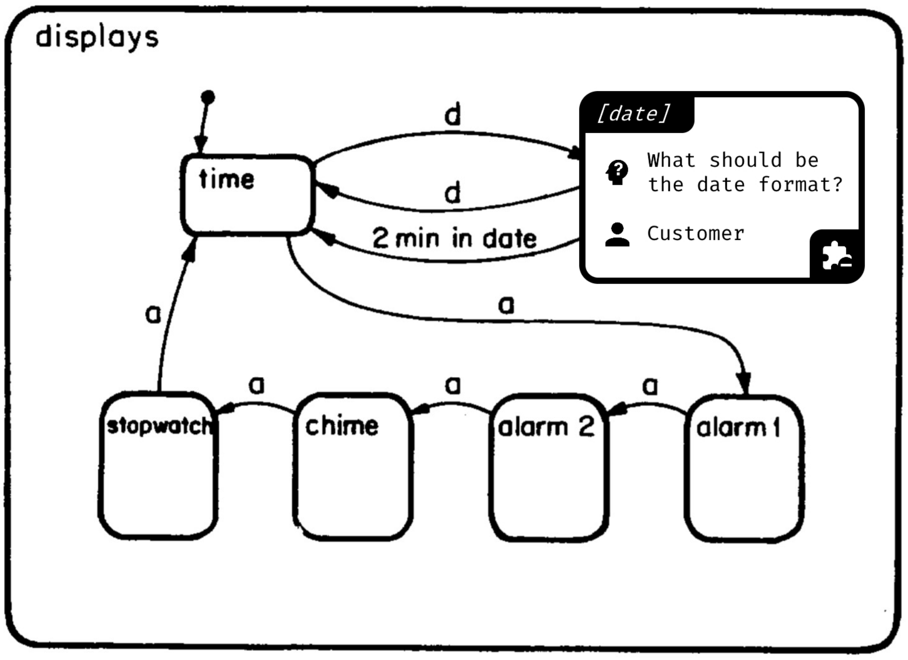


Figure 4.1: Statechart Including the State Hole "date"

date, which is indicated by the icon of a missing puzzle piece as well as the name put in brackets. As explained in Section 2.3.1, typed holes are labels that serve as questions to the compiler. Not targeted at compilers, but at other participants in the software development process, holes in statecharts convey way more information. A hole without any context is useless, thus the most important information is the reason for the creation of a hole. In this example, somebody in the software development process (probably a developer or requirements analyst) was unsure about the date format that should be used while presenting the date on the watch. Intended as a tool simplifying communication, the role of a person intended to respond (customer in this example) is specified as well. Summarizing this description the creator of this hole might have thought: "I already know there should be a state called date in the behavior of the watch, but I am unsure about the date format to display, thus I am asking the customer (or a product manager acting as the customer) for clarification."

By combining statecharts and hole-driven development it seems possible to

- *specify requirements* using a visual formalism (Leveson et al., 1991),
- create a *shared model of understanding* (see Section 3.2),
- that can be directly *executed* (Harel, 1987),
- and is *aligned to the thought process* of developing software (see Sections 2.2.2 and 3.1),
- while *improving communication* in the software development process (see Section 2.1.3).

Based on their conducted research, Visser and Hoc (1990) identified five requirements for tools that assist the design activity. Does the combination of statecharts and hole-driven development adhere to these requirements?

### 4.2.1 Requirement 1: Assisting the Management of Working Memory

The cognitive system *working memory* acts as a buffer between the sensory register and the long-term store, its capacity is limited and information is only held temporarily (Herczeg, 2018). Regarding the performance of the working memory, $7 \pm 2$ chunks can be stored for around $15 - 30$ seconds (Herczeg, 2018). As presented by Victor (2012), these limitations seriously restrict the understanding and mental simulation of a program's behavior, thus longing for interactive visual alternatives.

Based on the results of various studies, Dutke (1994) states that on principle visualizations support cognitive processes, while entailing the major risk of a mental overload. Such an example is presented in Fig. 4.2 (Harel, 1987). Despite the fact that the behavior of the full system is visible at one glance, this statechart is impractical to use in this form (Harel, 1987). It lost a model's character of abstraction. For getting an overview of the full system it is not abstract enough and for understanding a certain part of the wristwatch there is too much noise (Dutke, 1994).

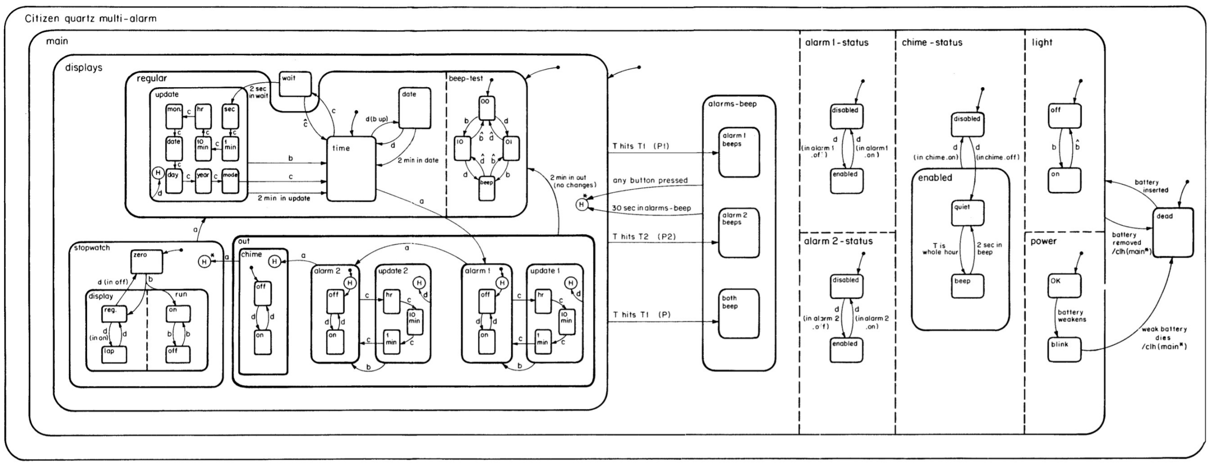


Figure 4.2: Full Statechart of a Digital Wristwatch Interface

Due to the fractal nature of statecharts (see Section 2.2.2) the various sub-states can be folded away and the same model can be visualized as shown in Fig. 4.3. It is important to note that the underlying model did not change, only its representation. As the limitations

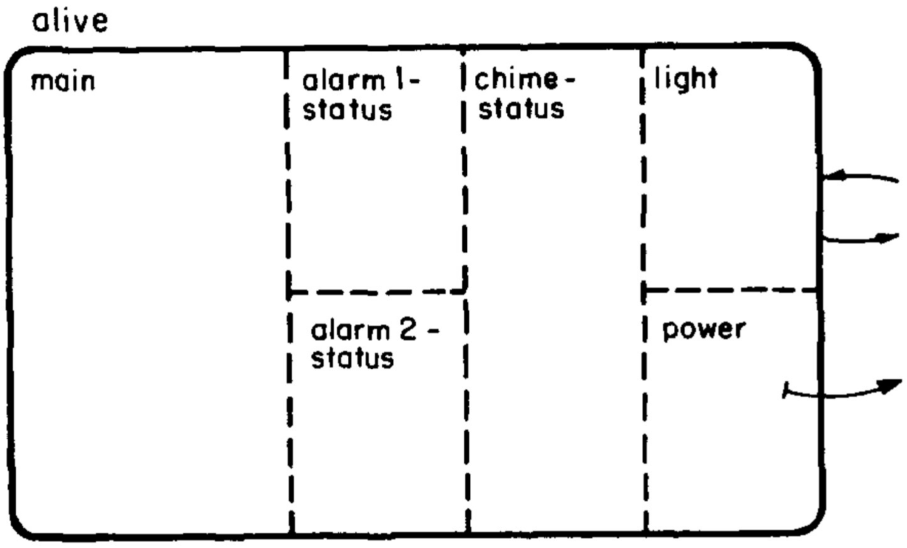


Figure 4.3: Simplified Statechart of a Digital Wristwatch Interface

caused by technology described in Harel (1987) were overcome, tools like the XState Visualizer[1] emerged, allowing developers to fold and unfold states at any time. This interactive way of creating different abstractions of the same model allows the selective use of working memory.

[1]https://xstate.js.org/viz/

### 4.2.2 Requirement 2: Enabling the Use of Libraries and Design Schemata

As mentioned by Visser and Hoc (1990) reusability is a major aspect for creating tools that assist system design. Reusing solutions of already solved problems is as important as shared knowledge about design schemata. Again the fractal nature of statecharts enables this property as defined by Visser and Hoc. Being able to create self-contained elements allows transferring these between projects or taking over existing solutions into the current project. Statechart implementations such as XState[2] or Statecharts.NET[3] simply reuse the infrastructure of the programming language they are written in. Package managers such as npm[4] or NuGet[5] enable the creation and distribution of packaged source code, thus offering an easy concept of reusability.

### 4.2.3 Requirement 3: Assisting the Articulation of Top-Down and Bottom-Up Components

Regarding problem decomposition strategies Visser and Hoc (1990) differentiate between the top-down and the bottom-up strategy. Programmers decomposing a problem top-down start at the root node of the solution tree and descend down against more concrete solutions, never going back up. As discovered by Brooks (1977), there are only rare cases where a clear top-down strategy really works. In these cases the programmers have to be experts and possess detailed knowledge in the problem domain (Dutke, 1994). Bottom-up strategies are defined by starting with the solution to small specific problems and creating the design in the process of solving more general problems. This approach to problem decomposition is usually taken by beginners and results in a worse software architecture than a well-thought-through top-down approach (Visser & Hoc, 1990). In reality a combination of both strategies is used where stepping back up in the solution tree followed by descending down again is called “backtracking”.

As suggested by Visser and Hoc (1990), programming languages shall support different levels of analysis and various flows of development, as the human brain works best not being forced to comply to one specific strategy. Already explained in Section 2.2.2, the fractal nature of statecharts enables programmers to nest and refine states as they create their solution tree. The statechart model directly maps to the top-down strategy (refining states)

[2]https://xstate.js.org/
[3]https://github.com/bemayr/Statecharts.NET/
[4]https://www.npmjs.com/
[5]https://www.nuget.org/

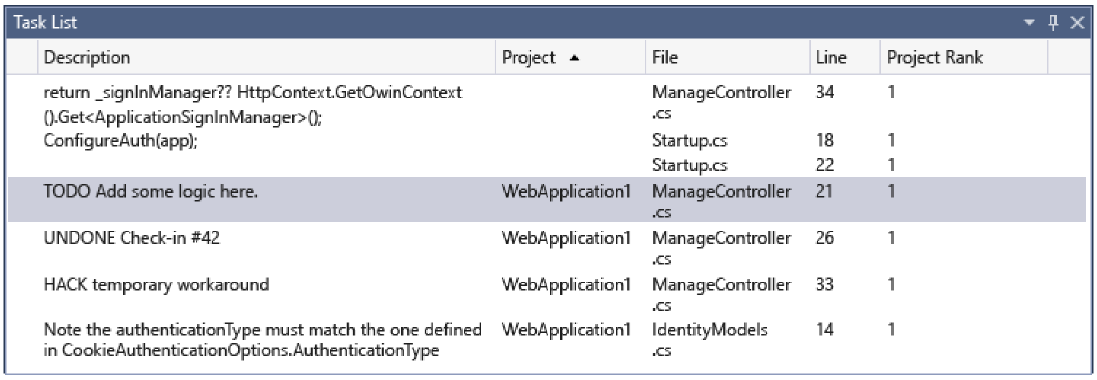
and the bottom-up strategy (nesting states), thus assisting system design as intended by Visser and Hoc (1990).

### 4.2.4 Requirement 4: Enabling Prospective Strategies

The strategy of prospective declarative problem decomposition helps at solving problems "where the complex structure of the input files and the relationships between them introduce strong and complex constraints on the program structure" (Visser & Hoc, 1990, p. 243). In contrast to the prospective procedural strategy where statements are written according to a mental execution strategy, the problems are too complex to be guided by already available procedures. This type of problem is called "ill-structured" (Visser & Hoc, 1990, p. 236).

For these kinds of problems it is especially important to provide tools that enable decomposition, because initially the problem space is too large to be solved at once (Dutke, 1994). By providing the aforementioned assistance in top-down and bottom-up problem decomposition the concept of statecharts already provides help for solving complex problems. Enabling prospective strategies is achieved by the combination with hole-driven development. By introducing holes programmers can write themselves notes for future use that cannot be forgotten, because when trying to execute the statechart it will fail with an incomplete definition.

Fig. 4.4 shows the *Task List* of the integrated development environment Visual Studio 2019 (Hogensen & Jacobs, 2019). This concept can be applied to the combination of


Task List

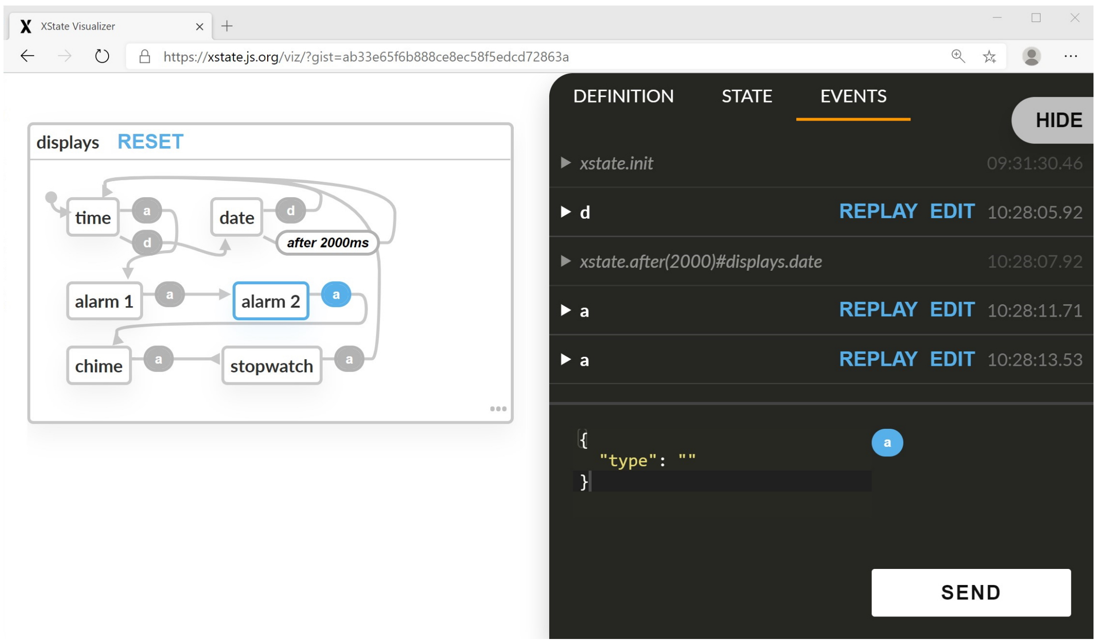
| Description | Project | File | Line | Project Rank |
|---|---|---|---|---|
| return _signInManager?? HttpContext.GetOwinContext ().Get<ApplicationSignInManager>(); | | ManageController .cs | 34 | 1 |
| ConfigureAuth(app); | | Startup.cs | 18 | 1 |
| | | Startup.cs | 22 | 1 |
| TODO Add some logic here. | WebApplication1 | ManageController .cs | 21 | 1 |
| UNDONE Check-in #42 | WebApplication1 | ManageController .cs | 26 | 1 |
| HACK temporary workaround | WebApplication1 | ManageController .cs | 33 | 1 |
| Note the authenticationType must match the one defined in CookieAuthenticationOptions.AuthenticationType | WebApplication1 | IdentityModels .cs | 14 | 1 |

Figure 4.4: Task List of a Modern Integrated Development Environment

statecharts and hole-driven development presenting an overview of all currently existing holes. Due to the additional information provided by holes as described in Chapter 4 this list can be targeted at specific participants in a software project keeping the mental overload as small as possible. Furthermore instead of showing the file location, the superstate of the hole can be displayed, enhancing the information presented with even more context. Instead of keeping in mind the tasks to do or using external tools, developers, system analysts, architects and other stakeholders in the software development process can be guided by the task list based on the holes in the statechart in a real *single* source of truth.

### 4.2.5 Requirement 5: Assisting Simulation

As described in Section 3.2 the simulation aspect of mental models is fundamental to their applicability. Statecharts, being a model as well, provide this same aspect of simulation. Fig. 4.5 shows this simulation aspect in practice, the used tool is called XState Visualizer[6]. Despite differences in notation, the statechart depicted in Fig. 4.5[7] on the left is identical



```
▶ xstate.init                                          09:31:30.46
▶ d                                     REPLAY  EDIT   10:28:05.92
▶ xstate.after(2000)#displays.date                     10:28:07.92
▶ a                                     REPLAY  EDIT   10:28:11.71
▶ a                                     REPLAY  EDIT   10:28:13.53
```

```
{
  "type": ""
}
```

Figure 4.5: Simulating Statecharts Using the XState Visualizer

to Fig. 2.2. On the right side there is an overview of the occurred events over time, the currently active state is highlighted directly in the statechart. Events can be sent by clicking on the event names directly in the statechart. Looking at the event log one can reconstruct why the currently active state is *alarm 2*. After initialization the statechart was in the state *time*. Sending the event *d* transitioned the statechart into state *date*, after 2 seconds it transitioned back into *time*. Sending the event *a* twice, it first transitioned

[6]https://xstate.js.org/viz/
[7]https://xstate.js.org/viz/?gist=ab33e65f6b888ce8ec58f5edcd72863a

into *alarm 1* and then continued into *alarm 2*. Creating this mental model of the system, one can simply play around with this interactive visualization, thus immensely reducing the cognitive effort of understanding and sharing the underlying mental model (Dutke, 1994).

# 5 Discussion

“One way to define ‘programming’ is as the process of transforming a mental plan of desired actions for a computer into a representation that can be understood by the computer.”

*Jean-Michel Hoc*

As Jean-Michel Hoc put it into words, a programmer’s main task is explanation. Before something can be explained, it must be understood deeply, also called the process of “theory building” as Naur (1985) observed, a predominantly psychological process. The major difficulty in interdisciplinary research is finding the sweet spot between the academic fields under research. This thesis is neither intended to generate groundbreaking findings in psychology nor in software engineering. Much more it should be a propositional thesis; it should be food for thought, proposing ideas that may lead to new fields of research.

Sheldon and Kim (2000, p. 10) put the author’s intention for starting the research on the psychological aspects of statecharts into words: ”Consequently, this approach can help to avoid the waste problem that results in redevelopment effort from incorrectly specified products.” The findings in the thesis definitely highlight the similarities between mental models and statecharts extended by hole-driven development. Whether the ideas presented in this thesis are applicable to reality needs to be investigated in future research.

One might ask why statecharts are not standard in reactive application development. Harel (2007) himself reflected on this topic concluding that the official semantics of statecharts were specified way too late which lead to the development of conflicting semantics that fractured the community. Additionally, unsuitable hardware, a lack of tools and learning resources might be the reason for the low adoption of such a promising concept, indicating the irony that a model suitable for transferring mental models failed at transferring itself. Breen (2004) published a critical view on statecharts stating that they might be too complex, suggesting that some features could be removed without losing necessary expressiveness. Regarding the topic of requirements analysis, Glinz (2002) examined how statecharts could be used as requirements models stating that: ”I have concentrated on requirements models only. The applicability and usability of the proposed statechart variant for other purposes, in particular for architecture and detailed design remains to be investigated.” (Glinz, 2002, p. 5) The usability of statecharts might as well be a really interesting research topic especially with groundbreaking papers released long after the formalism of statecharts such as “The ‘Physics’ of Notations” (Moody, 2009).

Further research topics might be: utilizing the model aspect of statecharts to create real evolutionary prototypes (targeted at computer scientists), the psychology of thinking in states and how this is related to preventing failures (targeted at psychologists), or adding the concept of literate programming (conveying mental models even better, [Knuth (1984)]) to the ideas presented in this thesis.

# List of Figures

# Bibliography

Anonymous. (1967). What A Programmer Does. *The Forum*.

Aranda, J. (2011). Naur's "Programming as Theory Building". Retrieved May 21, 2020, from https://catenary.wordpress.com/2011/04/19/naurs-programming-as-theory-building/

Brady, E. (2017). *Type-Driven Development with Idris*. Shelter Island, NY: Manning Publications Co.

Breen, M. (2004). Statecharts: Some Critical Observations. Retrieved March 4, 2020, from https://mbreen.com/breenStatecharts.pdf

Brooks, R. (1977). Towards a Theory of the Cognitive Processes in Computer Programming. *International Journal of Man-Machine Studies*, *9*(6), 737–751. doi:10.1016/S0020-7373(77)80039-4

Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., & Tolbert, D. (2017). *Unified Modeling Language (UML) Version 2.5.1*. Object Management Group (OMG).

Curtis, B., & Walz, D. (1990). The Psychology of Programming in the Large: Team and Organizational Behaviour. In *Psychology of Programming* (pp. 253–270). doi:10.1016/B978-0-12-350772-3.50021-5

Dutke, S. (1994). *Mentale Modelle: Konstrukte des Wissens und Verstehens: Kognitionspsychologische Grundlagen für die Software-Ergonomie (Mental Models: Concepts of Knowledge and Understanding: Foundations of Cognitive Psychology Applied to Software Ergonomics; in German)*. Arbeit und Technik. Göttingen: Verlag für Angewandte Psychologie.

Fowler, M., & Highsmith, J. (2000). The Agile Manifesto. *Software Development*, *9*(8), 28–35.

Gamari, B. (2019). Haskell: Typed Holes. Retrieved June 3, 2020, from https://gitlab.haskell.org/ghc/ghc/-/wikis/holes

Glinz, M. (2002). Statecharts For Requirements Specification – as Simple as Possible, as Rich as Needed. In *Proceedings of the ICSE 2002 International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, Orlando.

Green, T. R. G. (1982). Pictures of Programs and Other Processes, Or How to Do Things With Lines. *Behaviour & Information Technology*, *1*(1), 3–36. doi:10.1080/01449298208914433

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, *8*(3), 231–274. doi:10.1016/0167-6423(87)90035-9

Harel, D. (2007). Statecharts in the Making: A Personal Account. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages - HOPL III* (pp. 5–1–5–43). doi:10.1145/1238844.1238849

Harel, D., & Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The Statemate Approach*. New York: McGraw-Hill.

Herczeg, M. (2018). *Software-Ergonomie (Software Ergonomics; in German)*. De Gruyter Studium. Boston: Walter de Gruyter.

Hoc, J.-M. (Ed.). (1990). *Psychology of Programming*. Computers and People Series. OCLC: 831351202. London: European Association of Cognitive Ergonomics.

Hogensen, G., & Jacobs, M. (2019). Use the Task List. Microsoft. Retrieved May 19, 2020, from https://docs.microsoft.com/en-us/visualstudio/ide/using-the-task-list

Horrocks, I. (1999). *Constructing the User Interface with Statecharts*. Harlow, England; Reading, Mass: Addison-Wesley.

Kitchenham, B., & Carn, R. (1990). Research and Practice: Software Design Methods and Tools. In *Psychology of Programming* (pp. 271–284). doi:10.1016/B978-0-12-350772-3.50022-7

Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, *27*(2), 97–111. doi:10.1093/comjnl/27.2.97

Leveson, N., Heimdahl, M., Hildreth, H., Reese, J., & Ortega, R. (1991). Experiences Using Statecharts for a System Requirements Specification. In *Proceedings of the Sixth International Workshop on Software Specification and Design* (pp. 31–41). doi:10.1109/IWSSD.1991.213079

Mayr, H. (2005). *Projekt Engineering: Ingenieurmäßige Softwareentwicklung in Projektgruppen (Project Engineering: Software Development in Groups; in German)* (2., neu bearb. Aufl.). OCLC: 181447981. München: Fachbuchverl. Leipzig im Carl Hanser Verl.

McCracken, D. D. (1957). *Digital Computer Programming*. New York: John Wiley & Sons.

Microsoft. (2020). NotImplementedException Class. Retrieved May 17, 2020, from https://docs.microsoft.com/en-us/dotnet/api/system.notimplementedexception

Moody, D. (2009). The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, *35*(6), 756–779. doi:10.1109/TSE.2009.67

Myers, B. A., & Ko, A. J. (2009). The Past, Present and Future of Programming in HCI, Winter Park, CO.

Naur, P. (1985). Programming as Theory Building. *Microprocessing and Microprogramming*, *15*(5), 253–261. doi:10.1016/0165-6074(85)90032-8

Nielsen, J. (2010). Mental Models. Retrieved May 28, 2020, from https://www.nngroup.com/articles/mental-models/

Ousterhout, J. (2018). *A Philosophy of Software Design*. OCLC: 1043552353. Palo Alto, CA: Yaknyam Press.

Rouse, W. B., & Morris, N. M. (1986). On Looking Into the Black Box: Prospects and Limits in the Search for Mental Models. *Psychological Bulletin*, *100*(3), 349–363. doi:10.1037/0033-2909.100.3.349

Ryle, G. (1984). *The Concept of Mind* (University of Chicago Press ed). Chicago: University of Chicago Pres.

Sajaniemi, J. (2008). Psychology of Programming: Looking Into Programmers’ Heads. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, *4*(1), 4–8. doi:10.17011/ht/urn.200804151349

Schraube, E. (2012). Das Ich und der Andere in der psychologischen Technikforschung (The I and the Other in Psychotechnological Research; in German). *Journal für Psychologie*, *20*(1), 1–25.

Sheldon, F. T., & Kim, H. Y. (2000). Software Requirements Specification and Analysis Using Zed and Statecharts.

Staltz, A. (2015). Unidirectional User Interface Architecture. Retrieved June 7, 2020, from https://staltz.com/unidirectional-user-interface-architectures.html

Victor, B. (2012). Inventing on Principle. Retrieved May 20, 2020, from https://vimeo.com/36579366

Visser, W., & Hoc, J.-M. (1990). Expert Software Design Strategies. In *Psychology of Programming* (pp. 235–249). doi:10.1016/B978-0-12-350772-3.50020-3

Weinberg, G. M. (1971). *The Psychology of Computer Programming*. Computer Science Series. OCLC: 216809. New York: Van Nostrand Reinhold.

universität innsbruck

## Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

| | |
|---|---|
| 21.06.2020 | [illegible] |
| Datum | Unterschrift |