Towards a Theory of Factors that Influence Text Comprehension of Code Documents

Patrick Rein, Marcel Taeumel, and Robert Hirschfeld

Abstract The design of domain-specific software systems can benefit from participatory design practices making domain experts and programmers equal, collaborating partners. The source code of such a system might be a viable communication artifact to mediate the perspectives of the two groups. However, source code written in a general-purpose programming language is often considered too difficult to comprehend for untrained readers. At the same time, it is yet unclear what makes general-purpose programming languages difficult to understand. Based on our previous study and related work from programming pedagogy and cognitive psychology, we develop an initial theory of factors that might influence the comprehensibility of source code documents by untrained readers. This theory covers factors stemming from the features of source code, factors related to the visual appearance of source code, and factors concerned with aspects independent of code documents. This chapter discusses and illustrates these potential factors and points out initial hypotheses about how these factors can influence comprehensibility.

1 Motivation: Code Documents for Participatory Design

Software can generate value in many domains whose experts are not necessarily programmers themselves. Thus, the evolution of software in domain-specific projects leads to a collaboration of domain experts and programmers. This is particularly important for software systems which are highly domain-specific, for example payroll accounting systems, or geographic information systems. *Participatory design* can serve as a framework for the collaboration between domain experts and programmers as it regards them as equal partners in the design of the software system (Asaro 2000).

P. Rein (✉) · M. Taeumel · R. Hirschfeld
Hasso Platter Institute, Potsdam, Germany
e-mail: patrick.rein@hpi.uni-potsdam.de; marcel.taeumel@hpi.uni-potsdam.de; robert.hirschfeld@hpi.uni-potsdam.de

C. Meinel, L. Leifer (eds.), *Design Thinking Research*, Understanding Innovation,
https://doi.org/10.1007/978-3-030-62037-0_14

Participatory design emphasizes mutuality, reciprocity, and mutual learning. In the described situation of domain-specific software development, experts can learn technical possibilities and constraints from programmers with regard to the software to be created. Programmers can learn from the domain experts the inner workings of the domain, its vocabulary, and its constraints. Eventually, such a collaboration of groups from both areas of expertise can yield the creation of something more valuable than the sum of its individual contributions (Asaro 2000; Muller 1108; Rein et al. 2020).

To facilitate the participatory design process, teams use various practices such as playing out situations in dramas, collaborative game design, and mock-ups. Part of the purpose of these practices is the creation of concrete artifacts representing a shared language between the different groups participating in the design process. These artifacts should improve the mutual understanding of each others perspectives and needs, while creating a sense of shared ownership of the language (Muller 1108; Ehn 1988).

We argue that the source code of a software system has the potential to serve as a useful concrete artifact representing a shared language in a team of programmers and domain experts (Rein et al. 2020). First of all, source code explicitly expresses all domain knowledge relevant for the behavior of the system. Further, source code can be open to different interpretations. For domain experts it can serve as a written out formal model of domain knowledge. The exact execution semantics of the code might not matter much, as long as the meaning of the domain knowledge is sufficiently clear. At the same time, for programmers source code serves as a static description of the dynamic behavior of a computer. It describes the mapping from domain knowledge to technical infrastructure such as user interface components, or hardware input and output. Bringing these two perspectives together is a major challenge for software development, therefore source code, which combines these two perspectives, is an interesting artifact for participatory design.

1.1 The Challenges of Code as a Communication Artifact

Formal descriptions of the behavior of software systems have previously been proposed and have been used in participatory design in software development teams (Kensing and Munk-Madsen 1993; Barrett and Oborn 2010; Evans 2004; Luebbe and Weske 2012). However, multidisciplinary teams use descriptions containing specialized representations, such as diagrams or domain-specific languages (DSL) instead of the actual source code. This makes the described behavior more accessible to the domain experts. At the same time, these specialized representations require extra effort as they increase the distance between the domain experts and the actual system description in source code. This distance has to be bridged either by developers mapping these descriptions to actual code or by additional infrastructure and tools which have to be maintained (for example a DSL compiler and a corresponding debugger). In contrast to that, the actual source code of the system is always available. Further, changes to the source code are directly executable and no additional infrastructure has to be maintained.

How can program source code be used as a frequent communication artifact when exploring (or discussing) domain-specific terms and rules, which can also be expressed as natural-language text?

While source code written in a general-purpose programming language is readily available as a communication artifact, it is currently often regarded as difficult to understand for non-trained readers. Many of the mentioned formalism designed to be accessible for non-programming readers are motivated by this assumption. In a previous study, we investigated whether this assumption holds for object-oriented programs in a domain with simple rules (Rein et al. 2020).

The results of our text comprehension study showed that inexperienced readers performed worse on a process description expressed in object-oriented code than they performed on the English text variant. At the same time, the effect size in our experiment was rather small. As this was only a first study on the topic, final conclusions cannot be drawn. However the small effect size is still surprising as one would expect a formal document format such as source code to be generally difficult for inexperienced readers to read. Based on previous work, we again see that the mere fact that code documents are written in a language with formal semantics does not directly result in incomprehensible documents (Nardi 1993). Other features of code seem to influence the comprehensibility of source code. Thus, our refined research question is:

Which features of code documents make them more "difficult" to understand than English texts for readers with little to no programming experience?

A detailed understanding of what actually makes code "difficult" to understand could help designers of future languages in targeting non-programming readers to make conscious design choices for or against language features. Existing observations of difficulties faced by novice programmers are not sufficient in this regard, as novice programmers aim to learn to program while our target group might not necessarily intend to do so.

1.2 Overview of the Theory and a First Example

In order to investigate the obstacles to understanding code documents written in general-purpose programming languages, we describe an initial set of factors that potentially influence how well readers can comprehend the content of these documents. Thereby, we aim to create a theory of which features of source code, used to express dynamic processes, are difficult for readers with no programming background to understand.

We argue that existing theories on program comprehension do not apply as they are mostly concerned with the comprehension process of trained programmers (Von Mayrhauser and Vans 1995). Further, even theories from programming pedagogy can only be applied to a limited extent, as they mostly deal with learners specifically trying to learn how to program (Robins et al. 2003). In contrast, we investigate situations in which readers have no prior experience and in which readers do not intend to learn programming. Further, we take inspiration from cognitive psychology research results on the process of reading (Rayner et al. 2012). Our theory, however, focuses on the results of that process and does not try to contribute to the existing theories of the cognitive processes happening during text or program comprehension.

When untrained readers encounter a source code document, they face content presented in an unfamiliar form (for an example see Listing 1). In order to try to understand the content, they have to overcome several "obstacles" at different levels: from strange formatting, to alien vocabulary, and unfamiliar semantics.

The underlying challenge is the representation of domain knowledge through programming languages. For untrained readers, the document is, in fact, written in an unknown language. The language might include English vocabulary, but the grammar and semantics of the language are different from the grammar and the semantics of natural languages. We, argue that this can be somewhat mitigated by programming languages as long as the grammar and semantics are similar to the grammar and semantics of natural languages. Readers can then use their knowledge of natural languages to try to understand the source code. However, even with a completely familiar grammar and semantics, source code remains a means for expressing technical knowledge. Thus, the domain knowledge might be encoded in technical descriptions or the description of domain knowledge might be mingled with technical vocabulary. Both make it more difficult for readers to find relevant domain knowledge. We describe these factors, all resulting directly from the features of source code, in Sect. 2.

While the described features are inherent to source code, untrained readers might not notice them at first but will first notice that source code also looks different from natural language text. Due to its inner structure, source code is formatted and styled differently. For example, indentation is often used to visualize the underlying structure of phrases in programming languages. This can result in source code documents in which no two consecutive lines have the same indentation. We describe the factors concerning the visual appearance of source code in Sect. 3.

Finally, the comprehensibility of a document is not a property of the document itself but of a particular document and a particular reader. The background and attitude of readers with regard to formal languages might influence the comprehensibility. For example, readers familiar with complex sets of production rules, for example chemical reaction formulas, might have less difficulty when trying to understand a program written in a rule-based logic programming language. We discuss factors that are independent off a particular code document in Sect. 4.

The resulting list of factors is by no means complete but serves as a starting point to generate initial hypotheses to test. We expect that new factors will come up during testing the initial hypotheses and that some of the initial factors will turn out to be irrelevant. Our initial list of factors is informed by related work on program comprehension, programming pedagogy, results of cognitive psychology research on reading, and the qualitative results of our previous study.

Before explaining each group, we will give an overview of how these groups relate to each other. We will also introduce a running example, which we will use to illustrate the different levels of factors whenever suitable. The example shows how a step in a conference registration process is described in source code of the programming language Smalltalk (Goldberg et al. 1983).

1.3 Running Example

The following example is one step in the registration process of a commercially used conference registration system one of the authors worked on. Both excerpts are from the material we used in the experiments to test some of the initial hypotheses (Rein et al. 2020).

The English text version of the process step reads as following:

> Fifth, the participant will select the workshop they want to attend. Therefore, the system first determines all workshops available for the participant to attend. A workshop is available if it has capacity left and if the workshop is open for the participant type of the participant. [...] The system asks the user to select a workshop from the set of available workshops.

This text describes the interactions between a user, called "the participant," and the registration system. The workshop registration step is only one of several steps in the registration process. The longer text from which this excerpt is taken also defines the relevant concepts such as participants, the conference, and why the workshop registration matters to the overall process. The ellipsis in the middle of the excerpt includes rules describing what defines whether a workshop has capacity left and whether a workshop is open for particular types of participants. These rules are omitted as they are also omitted in the source code excerpt below. This does not mean that the description in source code does not express these rules, but that they are not expressed in the excerpt used.

Listing 1 The example process step expressed in the Smalltalk programming language. The process step is expressed in a method called processStepFiveSelectWorkshop

```
1 ConferenceRegistrationProcess >>processStepFiveSelectWorkshop
2
3  | availableWorkshops |
4  availableWorkshops := self allWorkshops select: [:workshop |
5    workshop hasCapacityLeft and: [
6      workshop canBeAttendedBy: participant]].
7
8  participant setSelectedWorkshopTo: (
9    self askUserToChooseWorkshopFrom: availableWorkshops).
```

Now, compare the textual description above to the following excerpt in Listing 1 describing the same process step in the Smalltalk programming language (Goldberg et al. 1983).

We will briefly outline what some of the elements mean and how they map to the description in the English text. The first line tells us that we are looking at the class ConferenceRegistrationProcess and at the method processStepFiveSelectWorkshop . For the discussion of factors, it is sufficient to know that classes are collections of methods and methods include code. Further, when explaining the code, we will sometimes refer to *statements*. As a heuristic, statements in programming languages are what sentences are in natural languages. Line 4 to 6 are a statement that describes the rules defining which workshops are available. To get the list of all available workshops, we go through allWorkshops and select each one that hasCapacityLeft and canBeAttendedBy the participant . Finally, we say that we set the selected workshop property of the participant to the result of asking the user to choose a workshop from the availableWorkshops.

2 Factors Resulting from the Features of Source Code

By its very nature, source code is expressed in a formally defined language, such as the Smalltalk programming language (Goldberg et al. 1983). This alone might already explain why source code is difficult to comprehend to untrained readers: source code is written in a language they do not know. The meaning of a source code document depends largely on the semantics of the programming language, which is unknown to untrained readers, thus preventing them from comprehending the document. However, as our initial experiment has shown, even readers completely unfamiliar with programming can still comprehend large parts of a source code documents. So, missing knowledge about the underlying semantics of the programming language does not make source code completely incomprehensible but only hampers comprehension *to some degree*.

Further, our past experiment implies that other features of source code are also relevant. In a debriefing questionnaire, we asked for specific difficulties readers encountered. Besides general expressions of uncertainty with regard to the meaning of the document, participants mentioned specific aspects such as particular syntactic elements, as well as technical vocabulary such as “nil”.

Therefore, we shall take a closer look at features of source code documents that might influence comprehensibility. We identified four potential sources of difficulty: *discoverability of grammar*, *familiarity of semantics*, *decomposition versus linearization*, and *representation of domain knowledge*.

2.1 Discoverability of Grammar

The grammar of a programming language, just like the grammar of a natural language, determines which sequences of characters are valid phrases in the language and what role a word plays in a phrase. We argue that the *discoverability* of grammatical rules could potentially influence the comprehensibility of source code documents for untrained readers. In detail, we argue that the discoverability is determined by the *familiarity* or *explicitness* of symbols denoting special grammatical structures in code.

For programming languages, the strict adherence to the grammar is important, as the grammar is later used to determine how the code should be executed. The grammar of natural languages has a similar role. Research on the process of reading shows that one part of understanding the meaning of a natural language sentence is to associate individual words with their grammatical roles, such as subject and verb (Rayner et al. 2012). Assuming that untrained readers try to apply a similar process of reading to source code, readers would also try to use a grammar to assign roles to words in source code. However, the grammar of programming languages might be completely unfamiliar to them.

In addition to the above, the role of words or phrases in programming languages is often denoted by special symbols. These symbols can be whole words or special characters, among them punctuation characters. In our example above, the bars (“| ... |”) mark the beginning and the end of a list of temporary variables, in our case a list with only one variable called “availableWorkshops”. The usage of special words and symbols in programming language grammars can be located along a spectrum ranging from using only explicit words to using unfamiliar special characters.

The implicit meaning of special characters in general might make the grammar less discoverable. Thus, some programming languages avoid punctuation characters and use words instead, for example the language AppleScript (Cook 2007). We expect such *explicit* representations of the syntax to be more discoverable and in turn easier to comprehend than implicit representations. In Listing 2, we can see the difference between the two approaches by replacing some punctuation characters with explicit descriptions of what parts of the code mean:

Listing 2 A variant of the example method rewritten according to a grammar that makes the grammatical roles of elements more explicit

```
ConferenceRegistrationProcess >>processStepFiveSelectWorkshop

 temporary variables: availableWorkshops.
 set availableWorkshops to self allWorkshops select: do
   arguments: workshop
   workshop hasCapacityLeft and: [
     workshop canBeAttendedBy: participant]
 end.

 participant setSelectedWorkshopTo: (
   self askUserToChooseWorkshopFrom: availableWorkshops).
```

Somewhere between these two extremes is another option: to use punctuation characters from natural language. These punctuation characters are used to denote something similar to what they indicate in natural language. The programming language Smalltalk uses special characters in this way, as do many other programming languages. In our example above, we can see that the period is used to separate statements just as the period separates sentences from one another in natural language. We assume that this improves accessibility because untrained readers simply employ their familiar understanding of punctuation characters. In contrast, if we use unfamiliar special characters or common punctuation characters in unfamiliar ways, the grammar would become less discoverable and consequently the document less comprehensible. In replacing known characters with unusual ones, we can expect the document to become less comprehensible for untrained readers. This can be seen in Listing 3.

Listing 3 A variant of the example method rewritten according to a grammar that uses unfamiliar characters to denote grammatical roles

```
ConferenceRegistrationProcess >>processStepFiveSelectWorkshop

 / availableWorkshops /
 <availableWorkshops <— self:allWorkshops:select —>[:workshop /
   workshop:hasCapacityLeft:and—>[
     workshop:canBeAttendedBy—>participant]]>

 <participant:setSelectedWorkshopTo—>(
   self:askUserToChooseWorkshopFrom—>availableWorkshops)>
```

2.2 Familiarity of Semantics

The meaning of a statement in a programming language is formally defined by what happens in the computer when that statement is executed. So, in order to fully understand what a given statement in a programming language means, one needs to know the complete set of evaluation rules for that language. These evaluation rules are called the semantics of the programming language. We assume that two dimensions could influence text comprehension for untrained readers: *similarity of semantics to common sense*, and *number and combinations of evaluation rules used*.

The first aspect is again grounded in the process of reading (Rayner et al. 2012). In order to understand a sentence, readers of natural text first assign grammatical roles to words, then combine words into phrases structures, and finally combine these phrase structures into sentence structures[1] (Rayner et al. 2012). The grammar of the language and the lexical information for each word provide the information on the relation between the words in the sentence. These relations are then interpreted through the readers knowledge about the world.

For source code documents, untrained readers do not have any knowledge of the evaluation rules and thereby about the actual relations between words in the document. In order to still be able to understand the meaning of statements in the document, they might heuristically use their natural language grammar and lexical information. This in turn would mean that evaluation rules which are similar to common sense should make a document more accessible. Statements which make use of evaluation rules that are close to common sense could then be understood in the same way as natural language text. For example, for native English speakers, time should flow from top to bottom through the document, or names that have been defined at some point should be available from then on.

The following example snippet illustrates the spectrum between what might be regarded as common sense and what is special to programming language semantics.

```
availableWorkshops := self allWorkshops select: [:workshop |
  workshop isAvailable].
lastWorkshop := workshop.
```

In this snippet, the execution of statements happens from top to bottom, so time flows in the reading direction. After executing the first statement, the variable availableWorkshops contains all workshops which are currently available. We can use the variable availableWorkshops from now on. At the same time, the usage of the variable workshop in the assignment to lastWorkshop is not possible, as the name “workshop” is only valid within the block denoted by square brackets

[1]This is a simplified depiction of the full version of one of the theories on the process of reading. This part of the theory is sufficient for our argument.

("[...]"). However, for an untrained reader the name was used beforehand in this snippet, so it seems plausible to assume that it could be used further down. Consistently interpreting the scopes in which a name is valid is a task which can also be challenging for programmers when they do not know the programming language (Wilson et al. 2017).

Beyond the familiarity of the used evaluation rules, the number and combinations of rules used in a document might also influence the text comprehension.

2.3 Decomposed Versus Linearized

The way source code documents are structured is fundamentally different from how most natural language texts are structured. Natural language text is mostly written to be read *linearly*. In contrast, source code is decomposed into many small elements which are referenced from many different locations within the source code, similar to the way an encyclopedia is structured. We argue that this fundamental difference is a main obstacle for untrained readers who are used to consuming text in a linear fashion, from the beginning of the text to the end. In order to understand code, it has to be read by jumping from one element of the source code to another.

Code is decomposed to improve the maintainability of source code. The goal is to try to avoid any duplication so that every relevant domain concept is only expressed once within the document. At the same time, the code can also become less accessible for untrained readers. This is also indicated by related work on programming pedagogy[2] (Robins et al. 2003).

For example, to answer any questions about concrete scenarios based on our original example method, readers would need to first look up further information. The method processStepFiveSelectWorkshop describes the general steps to get the available workshops, but intentionally leaves out several details. To answer questions on whether one specific participant would be able to select a specific workshop, readers would need to know how canBeAttendedBy: is actually defined. To learn about its definition, they would have to scan the document and look for the definition of canBeAttendedBy: and read that definition.

Assuming that a linear version of our example would be more accessible, we could directly include the definitions of all other relevant methods directly within our example method. The resulting code might look similar to Listing 4.

[2]For example, a survey on studies on how to teach and learn programming found that object-oriented programming was difficult for novices because the program text was distributed across many small elements (Robins et al. 2003).

Listing 4 A linearized variant of the example method that includes the definitions of relevant other methods

```
1   ConferenceRegistrationProcess >>processStepFiveSelectWorkshop
2
3     | availableWorkshops |
4     availableWorkshops := self allWorkshops select: [:workshop |
5       workshop isUniversityWorkshop ifTrue: [capacity := 15].
6       workshop isCompanyWorkshop ifTrue: [capacity := 20].
7       workshopHasCapacityLeft := workshop attendance < capacity.
8       workshopCanBeAttendedByParticipant := workshop
            isCompanyWorkshop or: [
9         workshop isUniversityWorkshop and: [participant
              isLocalStudent]].
10      workshopHasCapacityLeft and: [
            workshopCanBeAttendedByParticipant]].
11
12    participant setSelectedWorkshopTo: (
13      self askUserToChooseWorkshopFrom: availableWorkshops).
```

Now, when answering questions about which participant can attend which workshop, readers do not have to refer to other methods. The phrases `workshop hasCapacityLeft` and `workshop canBeAttendedBy: participant` have been expanded with their definitions (see line 5 to 7 and line 8 to 9).

2.4 Representation of Domain Knowledge

Code always expresses the domain knowledge of the application domain in some way. At the same time, source code is also primarily a means to describe the behavior of a technical machine. Thus, source code necessarily intertwines the two aspects. We argue that two dimensions of this relationship have an influence on the comprehensibility of code: *how explicit the domain knowledge is expressed* and *the relative proportion of technical and domain vocabulary* in the document.

The first dimension influences comprehension as it determines how much the source code expresses logic of the domain versus how much it expresses the underlying operations of the computer. To create a software system, programmers inevitably have to map the domain knowledge to underlying operations of the execution environment at some point. At the same time, programming languages allow programmers to abstract from these underlying operations, for example by putting them in a separate method and giving the method a name which reflects the domain logic expressed through these underlying operations. We can then use this method wherever that domain logic is needed. Readers encountering the method name can understand what happens in terms of the domain and do not have to know which primitive operations are executed in the computer.

Listing 5 illustrates how a version of our example method would look with a somewhat less explicit description of the rules to determine which workshops are available. First of all, the explicit method hasCapacityLeft was removed as the method name describes knowledge of the domain. Second, the code expressing that we select specific workshops was replaced by a loop which iterates over the offsets in a primitive collection of numbers (line 5). The offset, called workshopIndex is used to look up the type of the workshop with that number in the mapping called workshopTypes. The type of the workshop itself is represented as a number which we compare to some known numbers (line 7 and 10).

Listing 5 A variant of the example method which represents domain knowledge through underlying data structures and operations, thereby making the expression of domain knowledge less explicit

```
1   ConferenceRegistrationProcess >>processStepFiveSelectWorkshop
2
3    | workshopsAvailable isCapacityLeft |
4    workshopsAvailable := Array new: self typesOfWorkshops size.
5    self workshopTypes indexDo: [:workshopIndex |
6      isCapacityLeft := false.
7      (self workshopTypes at: workshopIndex) = 1 ifTrue: [
8        isCapacityLeft := (self
9          workshopAttendances at: workshopIndex) < 15].
10     (self workshopTypes at: workshopIndex) = 2 ifTrue: [
11       isCapacityLeft := (self
12         workshopAttendances at: workshopIndex) < 20].
13     (isCapacityLeft and: [self workshop: workshopIndex
           canBeAttendedBy: self participant]) ifTrue: [
14         workshopsAvailable at: workshopIndex put: 1].
15
16   participant setSelectedWorkshopTo: (
17     self askUserToChooseWorkshopFrom: workshopsAvailable).
```

The overall structure looks similar to Listing 4. However, while Listing 4 still has method names which reflect knowledge about the name, such as isUniversityWorkshop, the code in Listing 5 no longer contains any method names with this vocabulary..

However, making much of the domain knowledge explicit and hiding all underlying operations might not be a guarantee for creating comprehensible source code. The domain logic could still be mixed with logic concerned with technical infrastructure, for example maintaining data structure or handling in- and output mechanisms such as user interface interactions. We assume that the more technical logic and vocabulary is intermixed with the domain logic, the less comprehensible the source code becomes. One argument for this is that the domain logic becomes less dense. Non-technical readers have to filter the technical details as noise to get to the actual domain knowledge in the document.

For example, Listing 6 shows how our example method would look if more technical logic were introduced. The most prominent part is visible at the bottom. The method askUserToChooseWorkshopFrom was removed and replaced with explicit handling of the user interface interactions (lines 10 to 17). The fact that the user is asked to choose a workshop is still expressed in these lines. However, the relevant words and phrases are intermixed with technical code, such as the unwrapping and converting of the result of the user interaction (lines 14 and 15).

Listing 6 A variant of the example method, which includes a mixture of domain vocabulary and technical vocabulary dealing with user interactions

```
1   ConferenceRegistrationProcess >>processSelectWorkshop
2
3    | availableWorkshops uiRequestResult chosenWorkshopIndex |
4    availableWorkshops := OrderedCollection new.
5    self allWorkshops do: [:workshop |
6      (workshop hasCapacityLeft and: [
7        workshop canBeAttendedBy: participant]) ifTrue: [
8          availableWorkshops add: workshop]].
9
10   uiRequestResult := UIManager default
11     chooseFrom: availableWorkshops
12     values: availableWorkshops
13     title: 'Please choose a workshop'.
14   chosenWorkshopIndex := (uiRequestResult at: #index)
15                          withBlanksTrimmed asNumber.
16   participant selectedWorkshop: (availableWorkshops
17     at: chosenWorkshopIndex).
```

As can be seen from a comparison of Listings 5 and 6, the two dimensions of how domain knowledge is represented are not completely orthogonal. When domain knowledge is encoded implicitly in technical data structures, the source code will necessarily contain the operations to work with these technical data structures and thereby *add noise* to the representation of domain knowledge.

3 Factors Related to Visual Appearance

As illustrated in the previous section, code is more structured than natural language text. Understanding the described behavior of the system fully, requires a complete understanding of the respective structure. Furthermore, the decomposed form of code, forces readers to often jump between sections in the code document. Thus, programmers often use visual cues to help them navigate the documents or recognize the structure of a statement more easily. In the following we will look at two aspects which determine the visual appearance of code documents, namely the *layout of the document* and the *formatting and styling*.

3.1 Document Layout

While, typically, source code is semantically decomposed into small elements, source code documents are still layouted just as natural language text is. Natural language documents linearly show one paragraph after another and source code shows one semantic unit after another, such as a class, method, function, or procedure. For example, the structure of the document containing our example method may look like Listing 7 (the content of the methods is omitted).

Listing 7 A shortened version of the source code document in which the example method is included, illustrating how the elements within a document might be ordered. The content of the methods is omitted

```
Object subclass: #ConferenceRegistration
  instanceVariableNames: 'participant'
ConferenceRegistration startRegistration [...]
ConferenceRegistration processStepOnePersonalDetails [...]
ConferenceRegistration processStepTwoEventType [...]
ConferenceRegistration processStepThreeParticipantType [...]
ConferenceRegistration processStepFourBookings [...]
ConferenceRegistration processStepFiveSelectWorkshop [...]
ConferenceRegistration limitOfWorkshopParticipants [...]
ConferenceRegistration limitOfParticipants [...]
ConferenceRegistration numberOfRegisteredParticipants [...]
ConferenceRegistration numberOfRegisteredBachelorStudents [...]
```

We argue that the *ordering of the semantic elements within a document*, might influence the comprehensibility for untrained readers.

For experienced programmers, an alphabetic ordering of the elements, or a grouping of methods according to a unifying topic, might ease navigation while jumping between methods. However, for untrained readers an ordering that corresponds to the likely navigation on the first reading might be more helpful. For example, the first method should be the most high-level method. All methods used by this high-level method should be listed below that high-level method. After these methods, all methods used by them are listed, and so on. Note, that this assumes that readers employ a top-down strategy when encountering the source code for the first time (Von Mayrhauser and Vans 1995). A reverse order might be used for the assumption that readers employ a bottom-up strategy (Von Mayrhauser and Vans 1995).

3.2 Formatting and Styling

The question as to how source code should be formatted has been discussed in the software engineering community for more than 40 years (Miara et al. 1983). We are more interested in only the distinguishing features of the code, independent of its presentation. However, as research on code presentation shows that some features can impact comprehension levels and speed, we would briefly like to discuss some of the common features: *syntax highlighting*, *indentation*, and *identifier styles*.

Syntax highlighting is a technique to enrich the visual information of source code. To highlight syntax elements, colors and text emphasis are added to parts of the source code that have special meaning. For example, a section of our original example might look like Listing 8 with some syntax highlighting added to emphasize the methods being sent.

Listing 8 A rendering of an excerpt from the example method with syntax highlighting emphasizing the names of methods used in the statement

```
availableWorkshops := self allWorkshops select: [:workshop |
   workshop hasCapacityLeft and: [
     workshop canBeAttendedBy: participant]].
```

Several empirical studies have investigated the effects of syntax highlighting. One study found that for reading source code in text books, syntax highlighting does not affect comprehension levels or speed (Beelders and Plessis 2016). Another study found that for novices trying to solve program comprehension tasks, based on small examples, syntax highlighting does significantly change the comprehension level (Hannebauer et al. 2018). While this does not imply that syntax highlighting does not help professional programmers or novices in writing code, it hints that the impact of syntax highlighting might be less important for our research question on factors influencing the comprehensibility of source code documents.

Another common question of code presentation is indentation. In all previous listings we have used the indentation of lines to show which statements belong together. For example, all lines within the example method were indented by one space and every statement within the square brackets of the first line was indented by at least three spaces. Indentation is said to improve the visual perception of such groups of statements. Without indentation it is more difficult to recognize these groups quickly. For example, the excerpt of Listing 8 would look like Listing 9 without indentation and coloring:

Listing 9 A rendering of an excerpt from the example method without indentation

```
availableWorkshops := self allWorkshops select: [:workshop |
workshop hasCapacityLeft and: [
workshop canBeAttendedBy: participant]].
```

In this version it is less obvious that the second and third line contribute to the list of available workshops in comparison to the original version. Correspondingly, one of the few studies on the topic found that indentation does indeed influence program comprehension (Miara et al. 1983). The effect on comprehension levels was rather small. For novices the effect was stronger than for professional participants. Further, participants reported a higher subjective difficulty of comprehending the source code when indentation was missing. Whether the impact of indentation on the comprehension levels of untrained readers is positive or negative remains unclear. While indentation might help discovering the hidden semantics of source code, it could also hinder the reading process by making the code visually more difficult to read linearly.

The final consideration with regard to formatting is the way names used in code are generated. Two major styles can be distinguished in contemporary programming languages: camel case and underscores. The following listing shows an example for each of the two styles:

```
canBeAttendedBy: "camel case"
can_be_attended_by: "underscores"
```

Program comprehension research shows that for experienced programmers and novices alike, there is no difference in correctness between the two styles (Sharif and Maletic 2010; Binkley et al. 2013). However, a one eye-tracking study found that the style using underscore results in some speed up (Sharif and Maletic 2010). The effect was larger for novices than it was for experienced programmers, indicating that with increased experience the influence weakens. While a similar effect might occur with untrained readers, we are mostly interested in comprehension levels not speed.

4 Factors Independent Off the Document

With this project, we aim to improve the code documents in order to improve comprehensibility. Thus, the factors presented so far focus on features of code documents directly. However, we also include factors beyond the features of code documents in our initial theory in order to inform future experiment setups. The first set of factors are concerned with the *readers* themselves. For example, beyond the basic reading and comprehension skill of readers, their past experience with any kind of formalism might influence how well they can deal with source code. The second set of factors captures the influence from the *application domain*. For example, a complex domain might make it even more difficult for readers to deal with the unknown format of source code.

4.1 Reader

While reading source code is different from reading natural language text, we suspect the general reading comprehension skill impacts how well a particular reader can comprehend the domain knowledge of a source code document.

The general comprehension skill level of readers is probably also influenced by whether they are native speakers of the language the code document is written in. While this seems like an obvious statement at first, it is important to keep in mind when considering source code. Most source code is written in English. Moreover, most programming languages use English words as keywords. Past studies have shown that this factor impacts comprehension by novice programmers (Guo 2018).

While we focus on untrained readers, a reader's past experience with document formats other than natural language text might influence how well the person can comprehend the code document. We assume that if readers have an educational background in a domain which makes heavy use of formal models, such as mathematics or systems theory, they might struggle less with the hidden semantics of the unknown programming language.

Beyond general and specific comprehension skills, the overall perception of the readers own assessment of their ability to understand a particular document format might influence the level of comprehension (Ashcraft 2002; Zhang et al. 2013).

4.2 Domain

Finally, for a given source code document, the level of comprehension a reader can achieve also depends on the domain described in the document. Two aspects of the domain might influence the comprehension level: the complexity of the domain and the familiarity with the domain.

The complexity of the content of a document influences how difficult it is for a reader to understand the document. Thus, more complex domain logic will make any kind of document harder to understand. However, complex domain logic might interact with the difficulty of comprehending the unknown format of source code for untrained readers. A more complex domain might in code result in more complex dynamic behavior, which on top of all the aforementioned challenges adds the requirement of being able to simulate that behavior in the readers mind. While this might influence future experiments, it can, in general, also not be solved, as the complexity of the domain is what we mainly want to express (Brooks Jr 1995). Reducing this complexity will subtract from what we initially wanted to express.

Finally, the familiarity with the domain has been shown to influence the program comprehension strategies used by professional programmers (Shaft and Vessey 1995). Programmers familiar with the application domain employ a top-down strategy to program comprehension, going from the high-level, domain-specific parts of the code to the more technical ones. Programmers who were not familiar with the application domain employed a bottom-up strategy, presumably going from what they know—this means from the low-level technical parts to the high-level, domain-specific parts. The study did not investigate whether the familiarity of the domain influenced comprehension levels. Nevertheless, we would argue that, for untrained readers, the difficulty of understanding an unfamiliar domain might interact with the difficulty of understanding the unusual format of source code and result in a decrease in overall comprehension.

5 Conclusion

Being able to read general-purpose source code, enables participatory design on the level of the fundamental definitions of the domain logic of a system. Enabling participatory design on this level is relevant in a variety of settings. For example, general software development can benefit when teams working on applications in domains with complex rules, or citizens might be able to participate in discussing how public administration processes are defined in open-source software. However, so far, the approach of language designers has been to provide representations of domain logic that were designed to be accessible to readers unfamiliar with source code. However, these representations require additional effort to keep them consistent with the actual source code. Consequently, we posed the research question of how to make general-purpose source code accessible to untrained readers.

This chapter did not answer this question, but instead described an initial theory of what might influence how well a reader can comprehend a source code document. In particular, we listed features of source code which might pose a challenge—namely the discoverability of the grammar, the familiarity of the semantics, whether code was presented in a decomposed or a linear form, and how explicit the domain knowledge was encoded. This theory is an initial proposal used to generate first hypotheses to be tested in experiments.

A more profound version of a theory would describe why untrained readers struggle with comprehending source code. Therefore helping future language and tool designers. General-purpose programming language designers can take the described obstacles into consideration and domain-specific language designers could even try to avoid these obstacles altogether.

References

Asaro, P. M. (2000). Transforming society by transforming technology: the science and politics of participatory design. *Accounting, Management and Information Technologies, 10*(4), 257–290.

Ashcraft, M. H. (2002). Math anxiety: Personal, educational, and cognitive consequences. *Current Directions in Psychological Science, 11*(5), 181–185.

Barrett, M., & Oborn, E. (2010). Boundary object use in cross-cultural software development teams. *Human Relations, 63*(8), 1199–1221.

Beelders, T., & Plessis, J. P. (2016). Syntax highlighting as an influencing factor when reading and comprehending source code. *Journal of Eye Movement Research, 9*, 2207–2219.

Binkley, D., Davis, M., Lawrie, D., Maletic, J.I., Morrell, C., Sharif, B. (2013). The impact of identifier style on effort and comprehension. *Empirical Software Engineering, 18*(2), 219–276. https://doi.org/10.1007/s10664-012-9201-4

Brooks Jr, F. P. (1995). The mythical man-month. Addison-Wesley Longman Publishing Co., Inc., USA ISBN: 0201835959

Cook, W. R. (2007). Applescript. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III* (pp. 1–1–1–21). , New York, NY: ACM. https://doi.org/10.1145/1238844.1238845

Ehn, P. (1988). *Work-oriented design of computer artifacts*. Ph.D. thesis, Arbetslivscentrum.

Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: The language and its implementation*. Boston, MA: Addison-Wesley Longman.

Guo, P. J. (2018). Non-native English speakers learning computer programming: Barriers, desires, and design opportunities. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (p. 396). New York: ACM.

Hannebauer, C., Hesenius, M., & Gruhn, V. (2018). Does syntax highlighting help programming novices? *Empirical Software Engineering, 23*(5), 2795–2828. https://doi.org/10.1007/s10664-017-9579-0

Kensing, F., & Munk-Madsen, A. (1993). Pd: Structure in the toolbox. *Communications of the ACM, 36*(6), 78–85. http://doi.acm.org/10.1145/153571.163278

Luebbe, A., & Weske, M. (2012). *When research meets practice: Tangible business process modeling at work* (pp. 211–229). Berlin: Springer. https://doi.org/10.1007/978-3-642-31991-4_12

Miara, R. J., Musselman, J. A., Navarro, J. A., & Shneiderman, B. (1983). Program indentation and comprehensibility. *Communications of the ACM, 26*(11), 861–867.

Muller, M. J. (2007). Participatory design: the third space in HCI. In *The human-computer interaction handbook* (pp. 1087–1108). Boca Raton: CRC Press.

Nardi, B. (1993). *A small matter of programming: perspectives on end user computing*. Cambridge, MA: MIT Press.

Rayner, K., Pollatsek, A., & Ashby Jr., C. (2012). *Psychology of reading*. Hove: Psychology Press. https://doi.org/10.4324/9780203155158

Rein, P., Taeumel, M., & Hirschfeld, R. (2020). *Towards empirical evidence on the comprehensibility of natural language versus programming language* (pp. 111–131). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-28960-7_7

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Shaft, T. M., & Vessey, I. (1995). The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research, 6*(3), 286–299.

Sharif, B., & Maletic, J. I. (2010). An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension* (pp. 196–205). Piscataway: IEEE.

Von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer, 28*(8), 44–55.

Wilson, P., Pombrio, J., & Krishnamurthi, S. (2017). Can we crowdsource language design? In *Proceedings of the Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* 2017. New York: ACM Press. https://doi.org/10.1145/3133850.3133863

Zhang, S., Schmader, T., & Hall, W. M. (2013). L'eggo my ego: Reducing the gender gap in math by unlinking the self from performance. *Self and Identity, 12*(4), 400–412.