(Author Version) Towards Empirical Evidence on the Comprehensibility of Natural Language versus Programming Language

Patrick Rein, Marcel Taeumel, and Robert Hirschfeld

Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany

Abstract In software design teams, communication between programmers and non-programming domain experts is an ongoing challenge. In this communication, source code documents could be a valuable artifact as they describe domain logic in an unambiguous way. Some programming languages, such as the Smalltalk programming language, try to make source code accessible. Its concise syntax and message-passing semantics are so close to basic English, that it is likely to appeal to even non-programming domain experts. However, the inherent obscurity of technical programming details still poses a significant burden for text comprehension. We conducted a code-reading study in form of a questionnaire through Amazon Mechanical Turk and SurveyMonkey. The results indicate that even in simple problem domains, a simple English text is more comprehensive than a simple Smalltalk program. Consequently, source code in its current text form should not be used as a reliable communication medium between programmers and (non-programming) domain experts.

1 Introduction

Software can generate value in many different domains whose experts are not necessarily programmers. Thus, the creation and maintenance of such software for domain-specific projects leads to collaboration of domain experts and programmers. Both parties benefit from frequent communication and knowledge exchange. On the one hand, experts need to articulate the details they expect programs to do such as convenient data collection, processing, and visualization. On the other hand, programmers need to articulate technical possibilities and challenges to guide (and sometimes constrain) the experts' expectations. Eventually, such a symbiosis of two professions can yield the creation of something more valuable than the sum of its individual contributions (see figure 1).

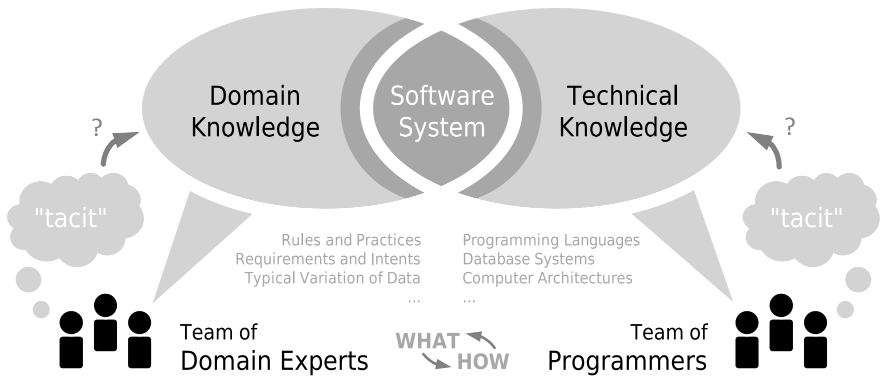


Fig. 1. An illustration of the way domain experts and programmers exchange knowledge when designing software systems.

In practice, programmers and domain experts try to establish a *shared language* to reduce the cognitive overhead in their discussions. Given an English-speaking team, such language might be regarded as "project-specific English" or "domain-specific English". The use of a shared vocabulary can reduce the number of comprehension errors and improve the team performance. This way, domain experts are *required* to express the relevant properties and rules in consistent terms. Programmers should apply these terms during code-authoring tasks. Given that a *software system* is being designed, we made two major observations: (1) all *words* in this language must have a *consistent meaning* and (2) *expressions* (or phrases) in this language have to be (made) *executable* for computers.

One approach to achieve such a shared communication medium is the use of (domain-specific) source code. Programmers are able to design domain-specific documents that still have meaning for the computer. Hidden technical layers can make the visible source code *look like* it was written in a (non-technical) domain language. Familiarity of (visual) form is often the basic argument for why domain experts might be able to think "in code". Eventually, there will always be complex, low-level details in the source code, but experts are not required to ever see them.

There are many domains where *natural-language text* plays an important role for capturing project-specific details. In such domains, the application of source code as a communication medium seems straightforward. While there might be technical details that distract non-programming experts, *domain-specific vocabulary* is discoverable in the form of high-level, executable code expressions [Evans2004]. This assumption leads to our research question:

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

Generally, when using source code as a communication medium, two opposing factors might affect the comprehensibility. The first factor is that domain-specific source code makes aspects of the domain knowledge explicit. For example, types of relevant objects of the domain, relationships between objects, or rules of the domain all have to be made explicit in order for source code to become executable. This explicit description of knowledge *reduces ambiguity* and creates a *detailed description of domain knowledge*. In contrast, even domain-specific source code documents inevitably include the formal syntax and the complex semantics of the programming language. For a reader with little to no programming experience, these might impose an *additional complexity burden* when trying to understand even simple code segments.

We suspect this additional complexity burden to outweigh the benefits of a less ambiguous description of domain knowledge, at least for non-programming readers. We see two implications if our claim holds. First, given the current means for designing domain-specific programming languages, there is still a major need for *external documentation* and thus other (maybe non-executable) forms of representing domain knowledge. Second, there is a significant value for language researchers to explore interactive means for programming beyond textual languages.

We present the results of a user study based on a controlled experiment on the assumption that even domain-specific, object-oriented source code poses comprehension challenges for non-programmers.

We chose the object-oriented paradigm because real-world domains can easily be modeled as object collaboration [Evans2004]. We chose Smalltalk [Goldberg1983] as a representative for an object-oriented programming language because its grammar is *close to basic English*. There, simple phrases consist of subject, predicate, and object. In Smalltalk, there are also objects that collaborate by sending messages. In comparison to English, the Smalltalk receiver of a message send is a *grammatical subject*, the Smalltalk message itself forms the *grammatical predicate*, and any message payload can be seen as *grammatical objects.* For example, the scenario "Every Friday, the postman delivers some mail to my address" translates to: "Date today dayOfWeek = #Friday ifTrue: [aPostman delivers: someMail to: myAddress]". In this way, Smalltalk can easily be applied to hide technical details such as request handling and resource management.

The hypothesis to test in this work reads as follows:

Given a problem domain with simple rules, people with little to no programming experience understand less details from a Smalltalk program than from an English text document.

Here, we define "simple" as (1) in the order of 10 domain rules, (2) in the order of 5 kinds of objects (or concepts), (3) less than 300 lines of Smalltalk code, and (4) less than 500 words English text. We base these figures on personal experiences from past projects. We conducted the experiment with 31 participants. As a result, we found that while participants can answer questions about scenarios expressed in source code documents, they answer more questions correctly when working with a corresponding natural language document.[1]

In section 2, we describe our experimental setup in detail by describing the layout, tasks, and the participants. Our user study is a questionnaire, which we carried out through Amazon Mechanical Turk (MTurk) and SurveyMonkey and whose implementation we describe in section 3. In section 4, we present and discuss our quantitative and qualitative results. In section 5 we give an overview of related work in terms of text and code comprehension. Finally, section 6 presents concluding thoughts and we outline future work.

2 Experimental Design

As our experimental design is not based on a previous design, we first describe our experimental layout in detail including the operationalizing of the relevant variables. We further describe the design of the scenarios used in the comprehension tasks, as well as the design of the used questionnaires. As we conducted the experiment through MTurk, we report on the measures we took to ensure that we got participants with the target background.

2.1 Experimental Layout and Operationalizing of Variables

We conducted a quantitative user study through a controlled experiment with a fixed setup [Robson2002]. Our hypothesis addresses the *difference in comprehension* for two forms of text documents: (1) natural language and (2) programming language. Thus, we had to *operationalize* the variables (a) text comprehension and (b) programming experience. Note that we target English-speaking readers with *little to no* programming background. This renders such an experimental design very challenging in terms of transparency and reproducibility. Consequently, we employed a *within-subject* layout to at least cope with variations among different participants and focus on mitigating carry-over effects between rounds for individual participants. We designed *two similar tasks*, namely a conference-registration process and a store-checkout process, whose order we counter-balanced, including the text-or-code treatments. On the downside, we require sets of four participants to equally serve all groups (as depicted in figure 2):

- Group A: Round 1 is Text (Registration), Round 2 is Code (Check-out)
- Group B: Round 1 is Text (Check-out), Round 2 is Code (Registration)
- Group C: Round 1 is Code (Registration), Round 2 is Text (Check-out)

[1] You can access the archived materials used for conducting the experiment and the unprocessed and pre-processed data from https://doi.org/10.5281/zenodo.2540989

- Group D: Round 1 is Code (Check-out), Round 2 is Text (Registration)

Having such a setup, we depend on proper *training* to familiarize all participants with our notions of *text document*, *English style, and programming style.* In addition to learnability, we do not address other carry-over effects explicitly.

One could argue that a participant's *motivation* might affect the upper time limit of a round. At the same time, we expect participants to unnecessarily rush through rounds, which might make a lower time limit more relevant. As we had no prior experience with similar experiments, we did not impose any time limits.

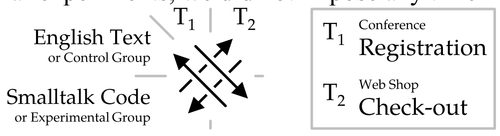


Fig. 2. The overall experiment layout showing all four groups.

To collect measurable results, we operationalized *text comprehension* through *answers in a questionnaire* about the document's content. Thus, we take the number of correct answers as a measure of the level of comprehension. For content presentation, we chose English as the natural language to attract many potential participants. We chose Smalltalk as the programming language, because of its small syntactic overhead [Goldberg1983]. Source code can be expressed in a concise fashion, sometimes almost matching their English-reading explanation.

To discard participants early in the process, we operationalize *programming experience* as the *number of lines of code written in the past* reported through *formal self-evaluation*. We expect this number to be very low or zero at best. Also, we query their experience with major programming paradigms such as the object-oriented paradigm [Wegner1987] or the functional paradigm [Abelson1996] on a five-point Likert Scale [Robson2002].

Finally, we want to stress our goal of *simplicity* in our experimental tasks (or scenarios). On the one hand, we favor simple English text. It should be neither literarily verbose nor artificially instructive. On the other hand, we favor simple program code in a clean style, closely resembling the vocabulary of the task's domain (i.e., conference registration or Web shop check-out). Consequently, our experimental design does not investigate isolated features of the documents but only the difference between a particular style of *natural-language text* and a particular style of *programming-language code*. To inform the creation of more detailed hypotheses, we collected *qualitative data* on the aspects of the documents the participants' thoughts and struggles in the debriefing phase.

2.2 Scenarios: Creation Process and Document Properties

Participants complete two rounds back-to-back. First, they answer questions about one scenario described in one language. Second, they answer questions about another (similar) scenario described in another (dissimilar) language. For each scenario, there are *ten questions* about the content and *two control questions* to verify participant authenticity, that is, to filter mechanical (or unserious) behavior. We do not impose time limits on the tasks.

Both scenarios are *representative* in the sense that they model existing, software-supported processes (or systems). In one scenario, we describe a conference-registration process, adapted from a real-world example one of the authors worked on. In the other scenario, we describe a shop-checkout and delivery-planning process, which has common characteristics of today's online-shopping processes.

Each scenario's domain guides the vocabulary, concepts, and rule of the descriptions (and questionnaires). The conference-registration process describes the online registration of a conference. It describes the interactions with prospective conference *visitors*, rules for deciding which extra *services* they can book, rules for deciding whether they can be registered or are put on a *waiting list*, and special cases for visitors with *invitation codes*. The shop checkout process describes the interactions with a customer of an online store for *books*, rules for deciding whether a customer is eligible for certain *shipping modes*, and a procedure for determining a *time of delivery*.

Both scenarios are similar yet different. They are similar in terms of their extent and structure, which addresses number of phrases (or code lines) and the choice of (English or Smalltalk) syntax. Yet, they are different considering their *source of complexity.* While, the registration process is simple with a complex set of rules for the visitor, the check-out process is more complex with a simple set of rules for the delivery slot. We discussed the comparability of tasks in section 4.4 as one source of threats to validity in our experimental setup.

To ensure similar structure, we first wrote the natural-language text and only then the corresponding code. After code authoring, we iterated over the English description, again, to align any phrases that turned out to be difficult to express in Smalltalk. This iterative procedure revealed unnecessary verbosity in the English text and artificial abstractions in the Smalltalk code. We formulated the questionnaire afterwards.

The style of writing can influence the comprehensibility of the documents. We found several rules to guide the application of the English language and the selection of Smalltalk features. While our main goal was to increase similarity between document representations, it helped us maintain a simple, yet representative, baseline. The rules for our English texts are:

- **Overview before detail:** Both scenarios describe processes. Thus, the reader should grasp the entire scope before diving into process-step details.
- **Paragraphs represent single steps and rules:** Readers might have to look up details in the description repeatedly as questions are answered. That look-up can be simplified if process steps and rules are at the granularity of text paragraphs.
- **Alternatives begin with "if":** There are conditions, which divide the processes into branches to be understood and recalled frequently. Thus, the reader should

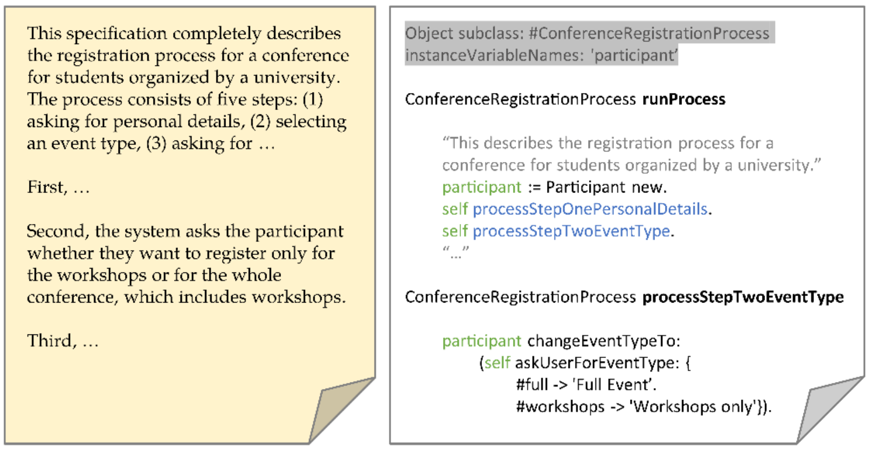
recognize such branches directly at the beginning of such sentences, not in the middle.

- **No synonyms:** Once decided on characteristic vocabulary for each scenario, the reader should not be bothered with extra complexity through synonyms. We kept a consistent vocabulary throughout process descriptions.
- **No text emphasis:** Being one characteristic difference between natural language and programming language, we reserve elaborate emphasis (such as colors, bold face, and italic face) for source code representation, not English text.

The rules for our Smalltalk code are:

- **Objects first:** Since Smalltalk favors the object-oriented programming paradigm, we model domain concepts with objects (and object classes) as much as possible.
- **Descriptive identifiers:** The vocabulary used to form Smalltalk expressions should reflect the domain as much as possible. This includes identifiers for variables and messages (or method names).
- **No explicit loops:** We want to avoid the cognitive effort of understanding loop constructs in code [Robins2003]. Smalltalk offers a concise alternative, similar to functional languages, with its collection protocol in the form of messages such as #select:, #count:, #firstThat:.
- **No recursion:** We want to keep the concept of message look-up as simple as possible. Thus, we want to avoid the idea of methods calling themselves repeatedly.
- **No meta-programming:** Smalltalk systems offer elaborate means to monitor and modify themselves. This powerful mechanism adds cognitive overhead that is not required for expressing simple domain rules. This includes the use closures in anonymous functions.
- **Descriptive data access:** For each data field, use an eponymous message (i.e. "*field")* to read but "*changeFieldTo:"* to write that field value. While this is against best practices in Smalltalk, pilot runs indicate that *"field:"* is too difficult to understand.
- **Minimize comments**: Comments reflect any additional information not represented in the code directly. None of the comments contain relevant facts for the questions.

Considering code formatting, we want to primarily guide document navigation in the experiment. We are aware that this is different from how programmers consume code in integrated programming environments. In particular, we use color to highlight structure and guide message look up. We highlight identifiers that store a value with a class described in the task in *green* and methods that can be looked up in *blue*. Additionally, code for each class is set in a section starting with a *distinct heading*. Each method is set as a distinct section (see figure 3). Overall, the code is formatted according to a standard set of idioms [Beck1996]. As an example, English text phrases correspond to Smalltalk code statements.



This specification completely describes the registration process for a conference for students organized by a university. The process consists of five steps: (1) asking for personal details, (2) selecting an event type, (3) asking for …

First, …

Second, the system asks the participant whether they want to register only for the workshops or for the whole conference, which includes workshops.

Third, …

```
Object subclass: #ConferenceRegistrationProcess
instanceVariableNames: 'participant'

ConferenceRegistrationProcess runProcess

    "This describes the registration process for a
    conference for students organized by a university."
    participant := Participant new.
    self processStepOnePersonalDetails.
    self processStepTwoEventType.
    "..."

ConferenceRegistrationProcess processStepTwoEventType

    participant changeEventTypeTo:
        (self askUserForEventType: {
            #full -> 'Full Event'.
            #workshops -> 'Workshops only'}).
```

Fig. 3. A comparison of a textual section and the corresponding source code section, including the actual type setting of the source code documents used during the experiment.

2.3 Questionnaires: Structure and Measurement

We designed questionnaires to gather *quantitative data* about the level of comprehension and *qualitative data* about general feedback on our experimental setup. There are *single-choice* questions on the scenario details and *free-text* questions on general concerns.

We distinguish two kinds of comprehension questions: (1) facts and (2) applications. First, questions about facts cover general properties and rules of the respective process. Such questions usually begin with "In which of the following circumstances can the following happen?" Participants can read these facts directly from the documents. Second, questions about applications require participants to assume particular properties for processes in action and derive implications. Such questions are usually like, "Given X, which of the following can happen?" or begin with "For this question, imagine the following scenario: …". Participants must therefore simulate the respective effect in their mind.

All comprehension questions are, *virtually,* single-choice questions. That is, even if there is more than one correct answer, participants are only required to pick one. We think that this is a fair trade-off for our hypothesis because multiple interpretations might be valid for vague (or abstract) descriptions. Note that for each question, we added two answers that express *generic* comprehension issues:

- Nobody can read this fact out of the given materials.
- I cannot answer the question.

Participants can therefore indicate that the provided materials do not provide enough information on some concern. We think that it is crucial in communication to be able to consciously express “I don’t know” and guess “Nobody can know”. This indicates that the current version of a document is not clear enough to serve as baseline for knowledge exchange between people. Such documents are usually created and maintained in an iterative manner. Mistakes can happen, waiting to be corrected.

For the (qualitative) feedback questions, we wanted to collect as much additional insight on our experiment and hypothesis as possible. Since we use MTurk, we cannot interview and debrief participants in person. Thus, an emerging structure of interview questions is not possible. Still, we encouraged participants to quote relevant passages from the provided documents to help us understand and interpret their concerns. We suggested the following topics for feedback: length of tasks, difficulty of questions, and obstacles to understanding.

In the end, a high level of understanding means a high number of correctly answered questions. We are aware that our selection of questions might not capture the entire knowledge base the documents encode. Yet, we did not consciously introduce irrelevant facts to distract or confuse participants.

2.4 Participants: Selection and Training

Our hypothesis addresses a wide range of people working in any field, but who do not have elaborate programming skills. Thus, we employed volunteer sampling by recruiting participants through MTurk to get a sample set of our target group. Since we cannot personally interview participants upfront, we installed several automated guards to ensure authenticity and high-quality results:

- **Basic engagement:** We required participants to have the “Amazon Mechanical Turk Master Worker” qualification, which indicates consistent high-quality work in a variety of tasks with a variety of contractors (or domains?).
- **Basic skills:** We required participants to have a U.S. bachelor’s degree, because we aim for scenarios that involve software developers and domain experts. We want to investigate their ways of knowledge exchange.
- **“Bot” detection:** We inserted control questions to check for the participant’s continuing attention. This might also reveal automatic (or unserious) behavior of participants who just want to collect the reward through MTurk very quickly.

We verify the authenticity of participants with *four control questions*, which we insert randomly throughout the questionnaires. Each question checks for common knowledge or basic involvement with the provided material. They are as follows:

- Which of the following kinds of processes does the document describe?
- Which of the following numbers is less than 100?
- What kind of item does the store sell?
- Which one of the following things is generally considered the tallest?

Even if participants are genuine, engaged, and educated, they cannot know what "little programming skills" means. They might accept this task erroneously, but with good intentions. Thus, we operationalized *programming experience* as a formal self-evaluation using questions to query written lines of code and familiarity with common programming paradigms. We rejected participants who reported *more than 1000 lines of code*. If such a number cannot be estimated, we resort to paradigm familiarity on a five-point Likert Scale: strongly agree, agree, neither agree or disagree, disagree, or strongly disagree. We provided the following statements to assess:

- I am familiar with the logic programming paradigm.
- I am familiar with the functional programming paradigm.
- I am familiar with the object-oriented programming paradigm.
- I am familiar with the Smalltalk programming language.

We think that familiarity with the Smalltalk language can also bias our results. Yet, we do not expect to find many Smalltalk programmers through Amazon Mechanical Turk. We rejected participants who expressed that they "strongly agreed" or "agreed" to any statement.

As we assume that working with formal models other than programming languages (such as construction processes and rule sets, chemical equations, statistical analysis, laws to some extent) might have a substantial influence on the performance on our tasks, we recorded the participants' experience with formal methods, and their professional and educational background. We did not reject participants based on this information.

With these filters in place, we recruited 35 participants with the expectation of achieving 30 genuine submissions. We rejected one participant due to level of programming experience, and ultimately recruited 36 participants, gaining 31 genuine submissions.

Since our experiment design entails potentially unfamiliar tasks, we provided *training tasks* to level relevant previous experience with reading comprehension tasks and thus mitigate carry-over effects from learning between rounds. This training consisted of an example scenario and questionnaire for the participants to practice. There was a shorter, but similar, textual description and *two* questions that also had the same structure as the questions used in the actual experiment.

As we explicitly aimed for readers with little to no programming experience, we did not provide training on the *semantics* of source code. However, we provided basic guidance on the *structure* of source code documents. Participants have to look up information, which might be straightforward in natural-language text but novel and uncommon in source code. So, we explained the role of *identifiers and method selectors* and our use of color for simplification. By example, we described this generic approach to find and connect information in code documents.

3 Experiment Procedure

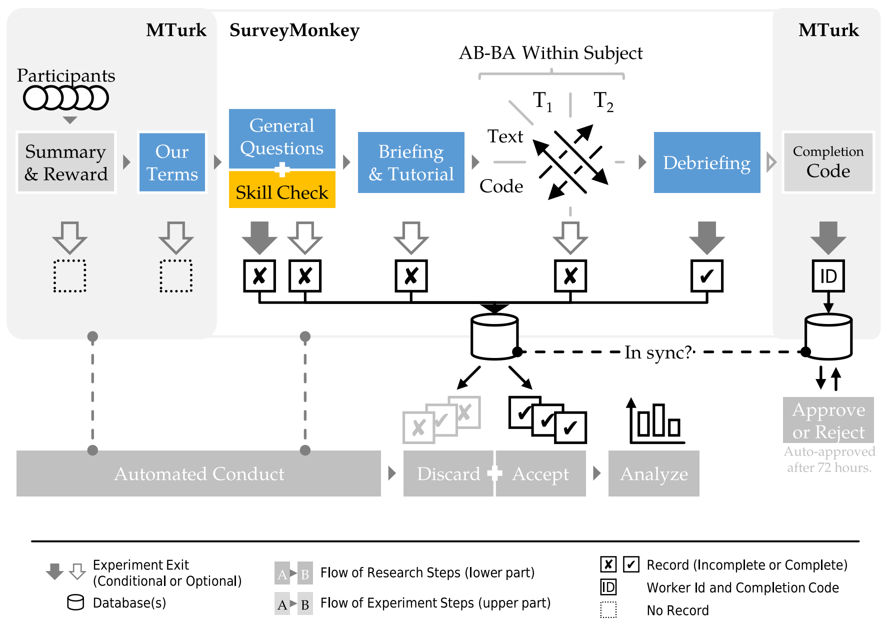


Fig. 4. An overview of the procedure of the experiment. The upper section shows the actual run of the study with the participants. The bottom section shows the overall process including pre-processing of data and the data analysis.

In this section, we describe the experiment procedure. We recruited participants through MTurk. The screening of participants and the actual experiment run was implemented using the SurveyMonkey service. After these automated processes, we manually filtered and analyzed the participants’ results.

3.1 Participant Management in Amazon Mechanical Turk

We posted the experiment as a task on MTurk, following guidelines for *academic requesters* written by the “Dynamo MTurk Community”[2]. These guidelines cover fairness and respect for participants including authenticity, privacy, payment, or time (or work) estimate. We advertised our experiment with the keywords “study” and “text comprehension” as follows:

[2] See https://web.archive.org/web/20190115201202/http://wiki.wearedynamo.org/index.php/Guidelines_for_Academic_Requesters and https://web.archive.org/web/20190115201354/http://wiki.wearedynamo.org/index.php/Basics_of_how_to_be_a_good_requester)

> “Answer questions about complex processes described in text and code (limited to participants with little to no programming experience) (~ 90 to 120 minutes)”

The estimated duration of the experiment is provided to give participants an impression of the expected time investment. We have no means to enforce this estimate. The participants were reimbursed US $15 on completion of the task or a fraction of this amount if they exited the task earlier. See section 2.4 for more details on our selection criteria. Note that we excluded all participants of (prior) pilot runs of the experiment through a project-specific MTurk qualification.

When previewing the MTurk task, participants were able to see the terms of the experiment. Unfortunately, we do not know how many participants saw the *initial* description including the reward or the preview of the terms. When they accepted the task, they were provided a web link to the SurveyMonkey questionnaire. At that point, our tracking of participants starts.

At the end of the SurveyMonkey questionnaire, participants got a completion code, which they entered on the MTurk task page to submit the task and eventually earn the reward.

3.2 Questionnaire via SurveyMonkey

We conducted the main part of the experiment through a SurveyMonkey questionnaire. We collected both *complete and incomplete* sets of answers because SurveyMonkey records all started sessions in the same database. Such incomplete records occur when participants exit the experiment early. They could decide to quit at any time or be disqualified by the process automatically during the initial screening (or skill-check) questions.

We designed the questionnaire for separate pages (for an overview see figure 4). The first page asked *general questions* about gender, age, and professional background. This page also included the questions related to participants’ *programming experience*. Depending on the answers, participants were automatically disqualified (see section 2.4), for example if someone was an expert in a certain programming paradigm. After this screening, participants received a *general briefing* on the structure of the questions and documents. That is, we suggested a primary reading method for both code and text. After this briefing, there was a tutorial page for training (see section 2.4). It contained example questions through which participants could familiarize themselves with the types of questions used throughout the experiment. This training concludes the preparation.

Participants then started with the actual experiment. At the beginning of each questionnaire, we added weblinks to the description of the current scenario as well as the general description on how to read (code) documents. We *disguised* these weblinks to prevent participants from peeking at the scenario’s other representation, which was code or text respectively. For example, one document was located at “…/code-fhbdz.html” and the other at “…/text-129d.html”. Further, we used the

SurveyMonkey mechanisms to implement the randomized assignment to participants of (a) document types, (b) scenarios, and (c) ordering. We further used SurveyMonkey to randomize the ordering of questions for each scenario as well as the ordering of single-choice options for each question.

The questionnaire concluded with a debriefing page, which included a question on general remarks as well as the completion code for MTurk to collect the reward.

3.3 Pre-processing the Results

As described above, we put several guards in place to ensure participant authenticity and thus genuine results. However, after the experimental conduct, the databases can include also records with obviously strange properties. Such records would unnecessarily bias statistical analyses and thus our interpretations. We employed the following steps to clean those records:

- **Synchronization with MTurk:** For all records on SurveyMonkey, we checked whether the provided MTurk worker identification number was listed in the list of workers who accepted the work on MTurk. We discarded all submissions whose identification numbers were not listed.
- **Repeated submissions:** We compared the IP addresses from which the sessions on SurveyMonkey were conducted. If the same IP address showed up in two records, we examined further for authenticity. That is, two people could share the same Internet connection—which would be acceptable. Otherwise, we discarded those submissions.
- **Incomplete submissions:** We discarded submissions that did not answer *all* mandatory questions correctly or incorrectly. Note that there were also feedback questions to gather qualitative data in the form of free-text fields, which were not mandatory.
- **Control questions:** We discarded submissions that did not answer all control questions *correctly*. We assume that such participants did not put the required effort into the experiment.

Note that we did not enforce time constraints. However, we noticed several records indicated low effort in terms of minutes spent. While we have no reason to remove them from statistical analyses now, we should think about ways to increase time spent and effort invested in follow-up studies.

4 Results and Discussion

After pre-processing the data, we analyzed the results with regard to the initial hypothesis. To ensure that our assumptions about the background of our participants were met, we determined several characteristics of our group of participants. Beyond the analysis of the quantitative results, we also summarized the results from the qualitative feedback we collected. Finally, we discussed threats to the internal and external validity of our results.

4.1 Participants' Background and Behavior

As we used volunteer sampling, our sample is not a true random sample. Thus, before describing the results of the comprehension tasks, we will first describe the characteristics of the participants. As a result of the described screening of participants and pre-processing of submitted questionnaires, we ended up with results from 31 participants. The participants are not equally distributed among groups (see table 1). This is since many participants signed up before a single participant had finished the questionnaire. Thus, we could not balance groups after participants dropped out or failed control questions.

Table 1. Number of participants per experimental group.

Group	A	B	C	D
Number of participants	8	9	5	9

Regarding the programming experience of the participants, most participants self-reported that they had never written any source code (see table 2). Three of the five participants who reported that they wrote some code (5-1000 lines of code) stated that they wrote small scripts to configure user interface scripting systems. The other two participants did not state what kind of code they created.

Table 2. The number of participants who reported a certain number of lines of code they wrote in the past.

Reported LOC	0	5	10	100	1000
Number of participants	26	1	2	1	1

Except for one, all participants either selected “disagree” or “strongly disagree” for any of the statements about their familiarity with any of the programming paradigms, the familiarity with the Smalltalk programming language, or their usage of formal methods. One participant selected “neither agree nor disagree” for two of these questions.

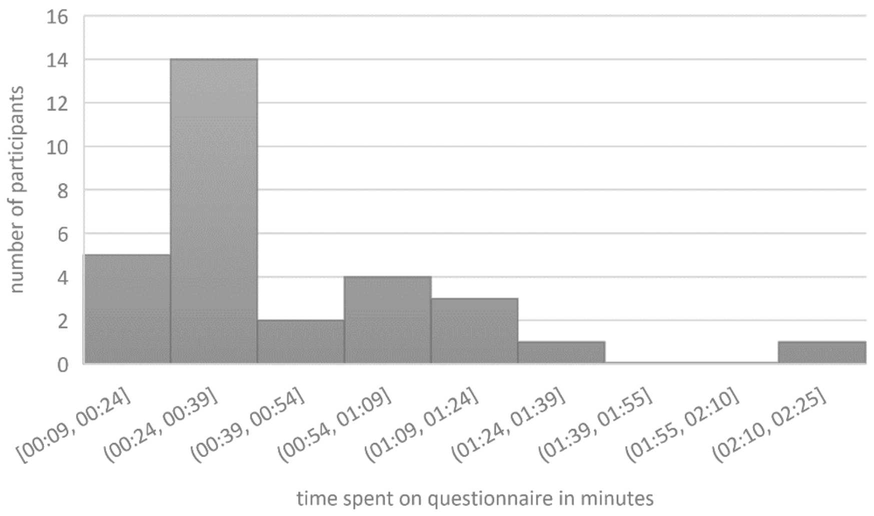


Fig. 5. Histogram of the time spent on the questionnaire excluding an outlier of 22h due to technical difficulties of the participant. Each compartment spans an interval of 15 minutes.

Finally, we determined the time spent on the questionnaire by taking the difference between the time the SurveyMonkey session was started and the submission of the last page of the questionnaire (see figure 5). Most participants spent 24 to 39 minutes on the questionnaire. Generally, most participants stayed under 99 minutes in total (excluding one participant whose recorded time was 22 hours and 58 minutes which was due to a technical issue on the side of the participant). As we intended participants to spent between 60 and 120 minutes on the task, the time participants actually spent on the task was lower than we intended.

4.2 Analysis of Quantitative Results

The primary goal of the experiment was to determine whether the format of the document influenced comprehensibility (see figure 7 for an overview of the results). As we employed a within-subject design, we used a paired t-test for comparing the results. Therefore, we determined the mean of the differences between the scores for the Smalltalk and the English language document of each participant (see figure 6).

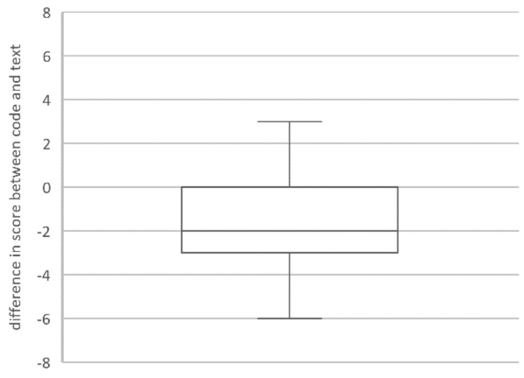


Fig. 6. A box plot of the distribution of the score differences for each subject between the code and the text document.

The differences were normally distributed, as assessed by the Shapiro-Wilk's test ($p = 0.197$). The mean difference is -1.42 (standard deviation 1.747) and the difference between the scores is significantly different from zero ($t_{31}=4.524$, $p < 0.001$). This difference means that on average participants scored fewer points on questionnaires about a Smalltalk document than they did on questionnaires about a text document.

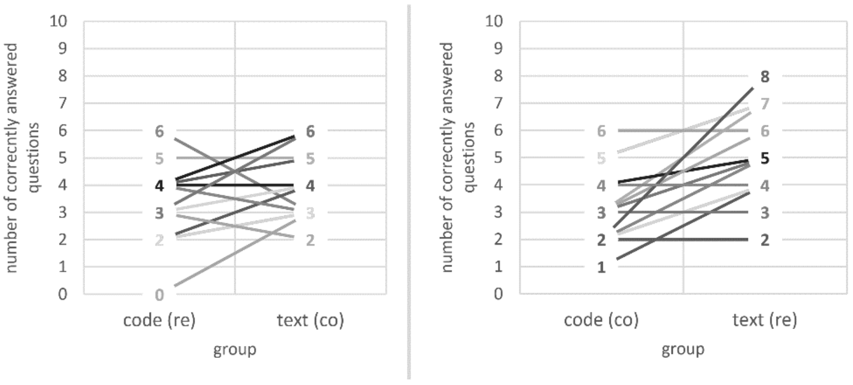


Fig. 7. Individual results for each participant between the questions answered on code and the questions answered on a text document. The two diagrams show the results for the respective combinations of document formats and scenarios (re = registration process, co = shop checkout).

Further, we designed the scenarios and the corresponding questionnaires to be of equal difficulty, to prevent any differences in difficulty from potentially influencing the results. To check this assumption, we also analyzed the difference between the scores for the conference registration and the shop checkout scenario. Again, the differences were normally distributed, as assessed by Shapiro-Wilk's test ($p = 0.356$). The mean difference between the two scenarios is not significantly different from zero ($t_{31}=1.656$; $p = 0.108$).

At the same time, the plots of the two groups show that the groups do in fact differ. For example, the group that worked on the registration process expressed in code had a lower minimum score for the code task and a lower maximum score for the text ask. Further, this group had two outliers who performed better on the code document than on the text document.

Despite the statistically significant results, the results should be regarded as preliminary. A detailed analysis of variance would be required to clarify the influence of the scenarios onto the results. Additionally, the current analysis does not cover the influence from the time spent on the questionnaire, the ordering of scenarios, or any interactions between these variables.

4.3 Summary of Qualitative Feedback

Following the completion of each questionnaire about a scenario, we asked participants to describe any difficulties they encountered when working with the documents. We did not apply a rigorous analysis to the data but summarized some of the themes in the responses below. As the participants wrote these statements after they finished all questions on one scenario, the statements do not allow for any conclusions on the actual difficulties the participants encountered. Nevertheless, they can serve as a starting point for future programming language features or experiment setups.

One topic which was primarily mentioned with regard to working with code documents was that participants felt that some *statements in the code lacked relevant context*. One participant stated this quite explicitly:

> "[…] the apparent answer to the previously stated issue appears later in this code block, but not in direct proximity to it, which may result in transient confusion."

The confusion was related to the meaning of the value of a variable in the conference registration process. Although the value of the variable was "#registered" the meaning of the value with regard to the registration process was unclear until the value was used later on to notify users that they were successfully registered. Another participant stated that they were unsure about the definition of a certain group of conference visitors:

> "[…] I also didn't understand if people who wanted to participate in the conference were also counted as going to the workshop and if that limit therefore applied to them (the 100 people limit). So, this line was challenging: 'self numberOfWorkshopParticipants >= self limitOfWorkshopParticipants' […]"

In order to understand the quoted code statement, one would have to look up the definition of the two identifiers "numberOfWorkshopParticipants" and "limitOfWorkshopParticipants". Both identifiers were defined in the code but in a different section of the code document. Again, the spatial distance between the statement and the corresponding detailed definitions might have caused the confusion.

A second topic with regard to the code documents was *difficulties with technical vocabulary or syntax*. While we intended to keep the code free from any unnecessary technical details, some details remained in the code. Some of these were mentioned as confusing or completely obscure:

> "[…] I did not understand what this meant: notNil so I could not interpret that code line at all. […]"

The "notNil" statement was also mentioned by other participants. However, only the quoted participant stated that this was a hindrance to understanding the surrounding line. Others simply noted that they were not sure what it meant. Another technical aspect of the code that was mentioned was the hash character, which is the syntax for denoting a symbol in Smalltalk (a piece of text with a unique object identity). An example in the source code is the symbol "#registered". This syntax was mentioned by one participant who quoted a section of code and stated that they were unsure about the meaning of the hash character:

> "[…] 'ifFalse: [| resultStatus |"
> resultStatus := #registered'but I don't understand what the # means. […]"

Notably, these two were the only technical aspects mentioned although the code did include a variety of technical vocabulary and syntax, such as the method name "->", square brackets, or curly braces.

Besides these topics primarily related to the code documents, one interesting theme with regard to the text documents was that participants would have preferred a different layout or representation of the text:

> "It was hard to apply the information in paragraph format. I had to break it down line by line. […]"
> "The challenge, for me, is that the information wasn't broken up into paragraphs or bullet points."

One participant even went so far as to suggested that a graphical representation of the process logic might help with understanding the underlying structure.

> "[…] I think in terms of my understanding and answering, some sort of flow chart or similar graphic might be the most useful. This would allow you to sort registrants methodically step by step based on attributes like local/visiting, code/no code, workshop-only/full […]"

4.4 Threats to Validity

While we consider the results of the analysis preliminary, we also want to point out the particular threats to the internal and external validity we identified with the current setup.

One *internal threat* is the fact that a large number of participants spend less than the intended time on the questionnaires. As the scenarios are complex and we can assume that the questions require concentration, spending less than 30 minutes on the tasks suggests that participants did not fully engage in the tasks. This matches the feedback on the overall study of one participant, who said: “I expected this to be a lot longer study. I would possibly include a progress bar […]”. Overall this might mean that we did not measure how well the documents might be comprehended but how well they might be skimmed or how quickly readers could find information relevant to a question.

Another *internal threat* results from conducting the experiment on MTurk. As we have no way to control the activities of participants while participating in the experiment, participants might have used external data sources for understanding the Smalltalk code, for example by searching for the meaning of method selectors. While we cannot control for it in general, future designs should incorporate a question at the end of the experiment as to whether participants used any information not contained in the experiment material.

As mentioned in the analysis of the questionnaire scores, the comparability of the difficulty of tasks might still be a challenge and thus be an *internal threat.* Additional analysis is required to determine the differences resulting from the scenarios. If the analysis shows that their difficulty differs, future designs will have to balance them better.

Despite two pilot runs, overlooked technical details in the scenario descriptions, such as the message “notNil” (see section 4.3), might have confused participants unintentionally and thus be another *internal threat*.

An *external threat* is the limitation of features of the Smalltalk programming language we used in the code documents. While Smalltalk code generally contains only few explicit loops, these might be necessary for some domain logic. At the same time, one could argue that if the domain logic does require an explicit loop, the complexity introduced by the loop was inherent to the domain. Beyond these situations, performance optimizations might require programmers to use more advanced language features in a code document.

Further, the minimal training on the semantics of source code might be unrealistic in practice and thus be an *external threat*. In situations in which domain experts regularly interact with programmers, one might argue that domain experts would quickly become proficient in the programming language.

5 Related Work

As the readability of source code affects not only source code as a communication medium but also the accessibility of source code for programming novices and the productivity of professional programmers, related work covers a variety of topics.

For example, a systematic series of controlled experiments showed that the different programming language syntax of existing programming languages has an impact on the accuracy of readers with no or little programming experience [Stefik2013]. The studies used a limited set of features, based on the set of features programming novices would first encounter when learning to program. The results of these studies were further used to inform the design of parts of the Quorum programming language.

Another study set out to investigate the impact of a domain-specific language on the performance of programmers [Ingibergsson2018]. The experiment design particularly targeted programming with a domain-specific language. The participants were all programmers who had some experience with the C++ programming language. While the results were inconclusive, the insights for designing code readability experiment could be used in future iterations of our setup.

Besides studies on the effect of textual languages, some studies examined the differences between textual and graphical representations of code. For example, one study investigated the impact of a textual and a graphical notation for regular expressions for readers with programming experience [Hollmann2017]. Regular expressions are a domain-specific language regularly used by programmers for pattern matching, for example in text segments. The study found that readers could answer questions on the expressions faster when working with the graphical notation.

Finally, a more general, related argument comes from literature on end-user programming [Nardi1993]. While programming languages are formal languages, studies on end-user programming suggest that the formal nature of code does not have to be an obstacle for understanding code documents. People regularly use formal languages in their everyday life without recognizing them as such. Examples for such languages are calculating sports statistics or sewing or knitting patterns. While the underlying languages are formal, they are not perceived by their users as such because they are primarily specific to a task that is familiar to its users.

Finally, a recent study suggests that the differences between code documents and text documents might become less distinct for experienced programmers [Floyd2017]. The study investigated whether the brain activity while reading source code is more similar to reading mathematical texts or to reading prose. While the results are preliminary, they indicate that with higher expertise in programming, the brain activity seems to become more similar to reading prose.

6 Summary and Conclusions

In the described study we set out to gather initial empirical evidence on the assumption that even domain-specific, object-oriented source code is insufficient to express domain logic in a way accessible to readers with little to no programming experience at all. Therefore, we devised a design for a controlled experiment through Amazon Mechanical Turk comparing the comprehensibility of documents about domain processes expressed in either Smalltalk source code or the English language. A first run of the experiment design resulted in data from 31 participants. Despite having no or very little programming experience, most participants were still able to answer several of questions based on the code documents. Nevertheless, the data provides statistically significant results showing that code is less comprehensible. However, this result should be regarded as preliminary as further post-hoc data analysis and more experiments are still required to clarify the influence of a potential difference in the difficulty of the scenarios.

The current experiment already hints that general-purpose programming languages might not yet be accessible enough. Beyond this, the described experiment setup could now be used for further experiments investigating how particular features of the code document or the natural language documents make the described domain knowledge accessible or not. These insights could then be used to design better languages and tools to make source code a useful artifact in the everyday communication between domain experts and programmers.

References

[Goldberg1983] A. Goldberg and D. Robson, Smalltalk-80: The language and its implementation. Addison-Wesley Longman Publishing Co., Inc., 1983.

[Robson2002] C. Robson, Real world research. 2nd edition. Blackwell Publishing, 2002

[Robins2003] A. V. Robins, Janet Rountree, and Nathan Rountree, Learning and teaching programming: a review and discussion. Computer Science Education 13, 2, 2003.

[Wegner1987] P. Wegner. Dimensions of object-based language design. In Proceedings of the International Conference on Object-oriented Programming, Systems, Languages & Applications, pages 168–182, ACM, 1987.

[Beck1996] K. Beck, Smalltalk best practice patterns. Prentice Hall, 1996.

[Raymond2010] R. P. L. Buse and W. R. Weimer. Learning a Metric for Code Readability. IEEE Transactions on Software Engineering 36, 4. IEEE, 2010,

[Stefik2013] A. Stefik and S. Siebert. An Empirical Investigation into Programming Language Syntax. Transactions on Computer Education 13, 4. IEEE, 2013

[Hollmann2017] N. Hollmann and S. Hanenberg. An Empirical Study on the Readability of Regular Expressions: Textual Versus Graphical. Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT), 2017 IEEE Working Conference on. IEEE, 2017.

[Ingibergsson2018] J. T. Mogensen Ingibergson, S. Hanenberg, J. Sunshine, U.P. Schultz. Experience report: studying the readability of a domain specific language. Proceedings of the 33rd Annual ACM Symposium on Applied Computing. ACM, 2018.

[Nardi1993] Bonnie A. Nardi, A small matter of programming: perspectives on end user computing. MIT press, 1993.

[Floyd2017] B. Floyd, T. Santander, and W. Weimer. Decoding the representation of code in the brain: An fMRI study of code review and expertise. Proceedings of the 39th International Conference on Software Engineering. IEEE Press, 2017.

[Evans2004] E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2004.

[Abelson1996] H. Abelson, G. J. Sussman, and J. Sussman. Structure and interpretation of computer programs. Justin Kelly, 1996.