Computer Code as a Medium for Human Communication: Are Programming Languages Improving?

Gilles Dubochet

École Polytechnique Fédérale de Lausanne

Abstract. Programming languages are not only useful to command computers, they also increasingly are a medium for human communication. I will use the framework of distributed cognition to discuss how knowledge is shared in a team of programmers and to show that computer code plays an important role in it. The resulting model of how programmers comprehend code suggests that common grounds play an important role in it. I propose two hypotheses concerning the means used by programmers to refer to common grounds from within their code. The hypotheses imply that modern languages, such as Scala, offer advantages as human communication mediums. I describe an experiment, using an eye-tracking device, that measures the performance of code comprehension. The hypotheses are tested by varying the degree of reference to common grounds.

Introduction

The original role of programming languages is that of a communication medium between a human and a computer. Today, the life span of software has increased, and programming teams have grown in size. As programmers need to communicate about software, computer code has also become an important *human* communication medium.

To describe computer codes, programmers often use words like "confused", "complicated", "to-the-point", "elegant", or "coherent", words not unlike those used for texts written in human languages. Of course, this *literary* quality of code is influenced by the experience and talent of their programmers. But contrary to human languages — where communication in English, Swahili or Chinese is about equally efficient — some programming languages seem superior in their potential to write "to-the-point", "elegant", or "coherent" codes. Such languages would obviously be superior mediums for human communication.

Improper collaboration has a destructive impact on software projects. It is widely reported in the professional literature that as much as one half of large projects fail. Most do not fail because technology is not mature, but because communication amongst stake holders, particularly programmers, is poor. The development process eventually stalls when the effort of sharing knowledge amongst programmers is equal to the amount of work all programers can achieve. A popular book by Brooks [3] describes how additional manpower can actually decrease the output of a team in such situations.

There is ample evidence — see, for example, Brooks [4] — that the maximum workable size of software projects has considerably increased in half a century. One amongst many factors may be that programming languages are improving in how suitable they are as mediums for human communication, thereby lowering the cost of collaboration.

In this article, I question whether relatively simple object-oriented languages, such as Java ,and richer object-functional languages, such as Scala [10], differ in their quality as mediums for human communication. To do so, I propose a model that emphasises the role of program code — and of programming languages — in the process of sharing knowledge in software projects. I describe two specific and testable factors that may impact the performance of a language performance in this role. These factors were tested in a controlled experiment, the results of which are reported in this article.

Section 1 presents related work on code comprehension as well as related work on applying the theoretical framework of distributed cognition to software development.

Section 2 describes a model of sharing knowledge in software projects based on the framework of distributed cognition. The model highlights the importance of computer code and of code comprehension in the process of sharing knowledge. It also emphasises the central role played by common grounds — knowledge that is shared by all members of a community. I formulate two hypotheses about what methods are used to reference common grounds from computer code.

- Code can refer to language constructs for which all programmers already have a mental model, lowering the cognitive load of comprehending it.
- Code can contain hints to its domain, simplifying the process of relating specific words or constructs in the code to concepts in the model, a process called "grounding".

Section 3 describes and section 4 analyses an experiment that tests these hypotheses. Code is modified to either resemble that of a simple object-oriented language (Java) or to utilise advanced constructs found in richer object-functional languages (Scala), as common grounds are referenced differently in these two forms. An eye-tracking device and questionnaires are used to measure variables related to code comprehension.

Section 5 discusses whether the result of the experiment support the model of code comprehension and whether they validate the hypotheses.

1 Related Work

Code comprehension has been recognised as a problem by the programming community for a long time. A 1989 article by Corbi [6] cites a study that concludes that "more than half of the programmer's task is in understanding the system." It goes on to postulate the existence of a "multileveled internal semantic structure to represent the program". Two methods of program comprehension are described, based on empirical research. Whilst not grounded in distributed cognition or in another cognitive theory, the intuitions described in Corbi's article are consistent with the model described in section 2.

Corritore and Wiedenbeck [7] further studied what mental representations are constructed during a code comprehension task, and compared the performance of procedural and object-oriented C/C++ code. Using questionnaires, their study differentiates between mental representations that are related to the domain and those that are related to the actual program, a distinction similar to that made in the present article. However, the study does not show a difference between procedural and object-oriented code in the mental representations built by experts.

Bednarik et al. [1] study the relation between gaze patterns and mental models. Whilst not comparing code comprehension in different languages, their concerns are homologous to those described in section 4. They report that a subject's code comprehension outcome that contains *info-high* information — comparable to the present article's "conceptual" outcomes — does not indicate correct comprehension, something that is also observed in section 4. They further observe that the proportional time spent reading certain families of code tokens does not correlate to the quality of mental models. That differs from the results reported in the present article, most likely because it uses a different model of token families.

Burkhardt et al. [5] refine the notion of mental model into distinct levels of cognitive representation. They report on a study which shows that tasks of different nature, such as documenting or modifying code, as well as different levels of expertise, lead to different levels of cognitive representation.

Bednarik and Tukiainen [2] use eye-tracking to analyse debugging with multiple representations. Beside code, subjects were presented with a visualisation of the problem domain and with debugging output. They report that, in later phases of debugging, experts rely more on code and on debugging output than novices, who heavily rely on visualisation. These results may provide insights into the actual process of building a mental model, something which was not discussed in the present paper.

Contrary to the previously-cited works, the present article relies on the framework of distributed cognition to ground its analysis. Distributed cognition [9, 11] has been proposed as a framework to analyse programming-related tasks, for example by Flor and Hutchins [8] in an analysis of a software enhancement task. They do not, however, detail the role of program code in the system. Wright et al. [14] propose to use the "resource model", an extension of distributed cognition, to analyse human-computer interaction. This model clarifies the distinction between "abstract information structures" and their representation. Wright's article does not discuss programming or code comprehension; it is manifest, however, that an analysis of such tasks benefits from making that distinction. The present article does not directly refer to the resource model, but the distinction between abstract information structures and their representation underpins it.

Below I report some experimental results on code comprehension within the framework of distributed cognition. The need for "[extreme programming] teams to maintain considerable common ground" is reported by Sharp and Robinson [12]. This result is in line with the model presented in section 2, although extreme programming may not be representative of all programming processes. Zayour and Lethbridge [15] discuss cognitive issues related to the design of software comprehension tools (reverse engineering). They claim that a tool must primarily aim at reducing cognitive load, instead of reducing the time required to perform individual tasks. Whilst this claim relates to comprehension tools, it is similar in nature to the first hypothesis that the present paper makes about programming languages. A similar notion permeates through Walenstein's investigation into the application of the resource model to the design of software comprehension tools (error tracking) [13].

To the author's knowledge, there is no previously published research that uses the framework of distributed cognition to describe the specific role played by program code in communication amongst members of a software development team.

2 A Distributed Cognition Model of Software Projects

A software project, whether large or small, is a knowledge-intensive endeavour. As a system, the project must memorise large amounts of knowledge, which is composed:

- of a model of the domain to which the software relates, including its vocabulary, concepts, et cetera (*domain model*),
- of the goals of the project, in term of reliability, performance, design, et cetera (*project goals*).

For the code of every component (object, function, algorithm, module, et cetera), it is furthermore composed:

- of a model of its design (*design model*),
- of a model of its purpose, of what it offers to the rest of the program, and under which conditions (*purpose model*).

Not all of this knowledge is in all programmer's memories. Programmers need knowledge about the design of the components that they change and about the purpose of the components that they use. They also require a limited knowledge of the domain model and of the goals. A large part of the knowledge is also, or only, stored in artefacts — code, documentation, et cetera —, which thus accomplish memory tasks on behalf of the project. Because memory is distributed, knowledge about the locations of knowledge (transactive memory) is further required. In such a system, artefacts simultaneously serve to carry knowledge from one programmer to another, and to store knowledge that isn't presently needed, for future retrieval. Natural language documents obviously serve as memory artefacts; in my experience, however, it is code itself that is the primary medium for storing knowledge. After all, programming languages are designed to describe the subtle and detailed properties typically relevant to programming. Furthermore, computer code necessarily exists for the entire software, something that is rarely true for other forms of distributed memory.

Performing even simple programming tasks on a component — such as adding a feature or fixing a bug — requires a fairly thorough model of its design and of its purpose. Despite that, such knowledge rarely stays in memory for long. It is usual for a programmer to be utterly unable to comprehend code he had written a few month before. This has probably to do with the sheer amount of knowledge that design models contain, but it is beyond the scope of this article to explore further why programmers do not hold to them for longer. On the other hand, knowledge about the domain model and the goals tend to remain memorised by programmers for long periods of time. An implication of this reality is that programmers very much depend upon being able to retrieve components' models from artefacts, particularly from code. When that is not possible, the difficulty of recreating fresh models often results in the component being discarded and rewritten.

The Process of Retrieving Models from Code

I will call the process of retrieving design models and purpose models from code: "code comprehension". Although there is no precise definition of the term "code comprehension" that I am aware of, its use in the literature usually seems consistent with this definition.

It is insufficient to retrieve only the knowledge contained in code to comprehend it. Rather, code comprehension is a process that requires putting together knowledge in the code with other, previous knowledge. Much of the previous knowledge that is required is project-specific — or domain-specifc — knowledge. To comprehend code, various forms of knowledge are required:

- a purpose model of the components that are used from within the code;
- a design model of the larger component, if the code is part of one;
- grounding in the domain model;
- the goal of the code within the project, which may help to justify design choices.

Knowledge that is not specific to the project will also be used, such as:

- emantic rules of the programming language — what happens when code calls a function, or how is a condition used to exit a loop;
- a purpose model of the components from the standard libraries that are used;
- recognition of the code of common behaviours in the form of "design patterns".

These forms of knowledge are "common grounds" that programmers use to comprehend code. Obviously, these grounds are common to different communities. The project knowledge is shared amongst members of the project, but parts of the domain model may also be shared by other domain experts. The other forms of knowledge are shared, depending on their nature, by the community of users of a language, of a library, of a programming style (object-oriented, functional), or by all programmers. Sometimes, common grounds are built by training or by explicit documentation. My experience is that much of the common grounds are formed by the process of code comprehension itself. Building the design and purpose models during code comprehension simultaneously validates and refines the domain model and the goals of the project. The same process is useful to construct knowledge about a programming language or about programming in general. Discussing further the processes of creating common grounds is beyond the scope of this article.

A corollary of the distributed cognition model of software projects outlined above is that code is very dependent on the common grounds to serve as an effective memory artefact. Even if all the necessary common grounds are available to the programmer, it is a non-trivial task to select, within the common grounds, the knowledge that is relevant to the code. This observation leads us to making two hypotheses related to how common grounds are used in code comprehension.

Hypothesis 1: Common Grounds Densify Code

Code is built out of pre-existing constructs for which models of their behaviour exists, and which are part of programmers' common grounds. These constructs, and the behaviour they

refer to, are like Lego bricks out of which the program is assembled. Code is "denser" if the constructs used to write code refer to models of more complex behaviour. The hypothesis is that code comprehension is improved with denser code for programmers whose common grounds are fitting.

The explanations for the success of the Java programming language are numerous. Most people do agree, however, that its standard library of reusable components has played a central role in it. These components provide behaviours that would otherwise have to be implemented using additional code. Furthermore, all Java programmers use these components and have a purpose model of their behaviour.

More generally, some languages or their libraries provide constructs with a higher cognitive content than others. Because these constructs contain additional cognitive meaning, fewer constructs are required to express a program of a given complexity, and code becomes shorter and denser. Of course, the models that a programer must eventually build do not become significantly simpler when code is denser — in the end, the cognitive content remains roughly the same.

Another aspect of high-cognitive-content constructs that may play a role in code comprehension is the literary opportunities that they offer. Languages with such constructs usually give more choice with respect to how an algorithm is described in code. Different constructs can be used to describe a similar behaviour, and programmers have to choose which one they use. That is not unlike choosing synonyms in human language. As an example, in Scala, an operation that is to be executed on many elements of a list can be coded by using many constructs such as "for-comprehensions" or higher-order functions. Amongst other things, these constructs differ in whether they emphasise the operation being carried out (higher-order functions) or the repetitive nature of the task (for-comprehensions). It is quite visible in many codes that programmers use this nuance as a way to structure better their code, as a way to have their code better "tell the story of their program".

Inexperienced programmers with a tenuous understanding of a language will not know how to decipher references to common grounds, and will probably not pick up the literary content of the code. In that sense, hypothesis 1 does not imply that all programmers will benefit from denser code. Instead, one would expect a decrease in the comprehension performance for those programmers who have not mastered the common grounds of the language.

Hypothesis 2: Hints Help Grounding in the Domain Model

Code has an imperfect relationship with the domain model, in the sense that, when considered by themselves, most codes imply a domain model which is too simple to allow for the comprehension of the code. Comprehension requires a programmer to flesh out the relationship between the code and the domain model. If the programmer already has a mental model of the domain, fleshing out the relationship is reduced to a task of *grounding* the code in the domain model. Furthermore, if the domain model is shared by programmers — if it is part of common grounds — code can be written so as to hint to the domain model and make grounding easier. The second hypothesis is that grounding hints improve code comprehension if the domain model is part of common grounds.

Depending on the programming language or programming style, grounding hints of different nature will be available.

- Programmers can freely name functions, values, data structures, et cetera. Some programmers keenly use this feature to ground their code, using complex names and the vocabulary of the domain model to describe, in English, the behaviour of a function, but many do not.
- Many languages allow data to be annotated with an expected type in the code. Types can be defined so as to correspond to the vocabulary and to the structure of the domain model — a concept the team refers to as "xyz" will be annotated using a type called "xyz". In some

languages, the coherence of type annotations can be checked automatically prior to executing a program.
- Error messages, which are part of the code, often serve as "negative" grounding hints.

Natural text comments inside code also play an important role. Modern languages have mostly concentrated on improving types. But even so, there is disagreement as to whether defining types that faithfully represent a domain model is worth their complexity. On the other hand, some have argued types to be useful as grounding hints even if they cannot be checked automatically.

3 Experiment

Each subject taking part in the experiment is asked to comprehend algorithms written in the Scala language. Scala is a modern object-functional language that allows denser code-writing styles than Java. To test the hypotheses described above, the coding style of the algorithms varies. We test the first hypothesis by presenting subjects with either code that uses dense Scala constructs or with code that only uses constructs available in Java. We test the second hypothesis by either using intermediate variable names relevant to the domain model — which serve as grounding hints — or by using meaningless variable names. Because of the relatively long duration of the comprehension task, the number of subjects is small (twelve); the experiment is not designed to yield statistically significant results on the dependent variable, namely the overall comprehension performance. Instead, it is primarily designed to observe process variables, particularly:

- the overall time spent looking at code during the comprehension task;
- the relative time spent looking at grounding hints and implementation code.

Method of Experiment

The subjects for the experiment are students in fields related to computer science or engineering from École Polytechnique Fédérale de Lausanne. The sample contains bachelor, master, and doctoral students. For the students I knew from teaching, they seemed fairly representative of the programming skills found in the student population.

Subjects are first presented with a short textual description of the concepts of relational algebra and with its vocabulary. In this experiment, relational algebra is the domain model against which code can be grounded.

Subjects are then presented with three algorithms, of varying complexities related to relational algebra. Each algorithm is coded by using one of three styles: S/G, D/G and D/U, which are described below. The same algorithms, in the same order, are presented to each subject, but the coding style for each algorithm changes from one subject to the other. For example, one subject is presented with algorithm 1 coded using style S/G and algorithm 2 using style D/U, whilst another subject is presented with algorithm 1 coded using style D/U and algorithm 2 using style S/G. Each subject will therefore be confronted with all three styles, although not in the same order and not for the same algorithms. The Scala language documentation is available during this phase of the experiment.

The three algorithms are implementations of relational algebra operators: (1) a natural join, (2) a left outer join, and (3) a cartesian product. These algorithms were chosen because the model of relational algebra involves non-trivial concepts and structures, and because the operators seemed awkward to comprehend without grounding them in the model. Table 1 lists the length — in number of lines and in code tokens (words and punctuation) — of the code for the three algorithms and for each coding style.

Independent Variables

The three different coding styles define two independent variables, which are related to the hypotheses.

	Lines of Code			Tokens		
Algo.	Sparse grounded	Dense ungrounded	Dense grounded	Sparse grounded	Dense ungrounded	Dense grounded
1	51	17	20	268	135	135
2	73	26	31	448	172	183
3	36	15	18	243	113	117

Table 1. The size, in lines of code and in tokens, for each algorithm in each style.

Variable for Hypothesis One The first independent variable is related to code density, and varies between the sparse (S/G) style and the two dense (D/G and D/U) styles. The sparse-grounded (S/G) style is typical for Java programs: it only uses simple iterative control structures, such as loops, and the entire variable state of the algorithm is represented as local, named variables. It represents the low-cognitive-content situation of the first hypothesis. As one can see in the table above, this style leads to long code, both in terms of lines and in terms of tokens. On the other hand, it visibly lays out the structure of the algorithm: the steps are easy to recognise and variable state is always explicit. The two dense styles use modern control structures to reduce the length of algorithms' code by a factor of more than two. They represent the high-cognitive-content situation of the first hypothesis. When possible, loops and variables are replaced by higher-order functions or "for-comprehensions". These constructs represent operations on lists at levels of abstraction higher than what loops and variables allow for. As an example, one can test if a condition is true for all elements in a list by simply using the "forall" higher-order function. Although the dense styles yield shorter code length than the sparse style, the algorithm implemented in all three styles are equivalent: the computer will eventually carry out almost the same calculation.

Variable for Hypothesis Two The second independent variable is related to the presence or absence of grounding hints. The two grounded styles (S/G and D/G) name elements (intermediate functions and variables) of the algorithm using words that are meaningful. They aim to ground the implementation within the domain model of relational algebra, using the same vocabulary that the subjects were presented with at the beginning of the experiment. A few intermediate names were added in the D/G style, when compared with the D/U style, to help grounding. The ungrounded style (D/U) concentrates on the raw implementation of the algorithm, using as little intermediate names as possible, and always using meaningless words.

Dependent and Process Variables

Once subjects are satisfied with their degree of comprehension, they proceed to write down the "programming contract" for each algorithm. A programming contract is a list of all pre-conditions that must be true for the algorithm to execute normally (for example: a certain list must not be empty), and of all post-conditions that will necessarily be true afterwards. During this phase, the code of the algorithms is still visible but the explanations on relational algebra and the Scala documentation are hidden. Finally, subjects are presented with a questionnaire on their level of confidence in their answers and on their skills in programming, in the Scala language and in relational algebra.

In order to collect data on the time spent comprehending each algorithm and on the time spent looking at different families of tokens, the experiment is conducted on a display capable of tracking gaze positions — a Tobii 1750 eye-tracking display, used without chin rest or bite bar. Gaze data is recorded during the entire comprehension task, including reading the description of relational algebra and describing the algorithms' contracts. Font size is set sufficiently large so as to allow connecting gaze to specific tokens in the code.

Prior to the experiment, subjects are told that they will participate in an experiment on "code comprehension using an eye tracker". All instructions for the experiment are provided in a written form and verbal interactions between subjects and the experimenter prior to the experiment are kept to a minimum. Each subject is compensated with twenty Swiss francs at the end of the experiment. The instructions make it very clear that the compensation is due irrespective of the way the experiment runs. All twelve subjects completed the task, most in between 45 minutes and one hour of time.

Method of Analysis

Dependent variables The contracts that the subjects have written for each algorithm are reviewed by the same experienced programmer who has written the algorithms, and assigned one outcome:

- failed, for evidently wrong contracts;
- serviceable, for contracts related purely to the algorithm's code. For example: "returns a list of merged mappings that are compatible; two maps are compatible if every common key contain identical elements" ("list", "map" and "key" are concepts related to the implementation);
- conceptual, for contracts placing the algorithm within its model. For example: "The relation contains all tuples that match common fields in the two relations" ("relation", "tuple", "field" are concepts related to relational algebra);
- unknown, for contracts that are unclear or too short to be placed in another category.

Process variables Raw gaze data from the eye-tracker are categorised into reading times for the following categories of elements:

- english sentences describing relational algebra;
- program code, by algorithm and style;
- text fields, were subjects describe contracts.

Program code is categorised further in token families:

- identifiers, for names identifying sub-functions or values of the algorithm;
- types;
- implementation, for any other code.

Only the last family of tokens contains useful information for comprehending the algorithm itself. The other two families ground the implementation in its model, either from the point of view of its correctness (types) or conceptually (identifiers, when in D/G and S/G style).

The reading times measured for different algorithms cannot be compared directly; the algorithms are of different complexities, and the first algorithm that a subject tries to comprehend can be expected to take longer. Therefore, all times are normalised by the average reading time for the algorithm they relate to. Normalised times for all algorithms are then compared as if all algorithms had the same complexity and as if no order effect was present.

Some gaze data that was collected during the experiment is not included in the analysis. Data for three subjects (nb. 4, 6, and 8) were rejected, based on the following criteria:

- different styles of the same algorithm are presented to a subject;
- gaze data is available for less than 50% of the duration of the experiment.

I do not expect rejected data to introduce a statistical bias, as rejection criteria are purely technical and are unlikely to be correlated with any other observed variable.

4 Analysis of Results

Results by subject

Table 2 shows the total duration of the experiment for each subject, and the proportion of that time for which gaze data is available. For each subject, it also shows the self-reported skill levels in programming, the Scala language, and in the model of relational algebra. Unavailable gaze

data are due either to the eye tracker not recognising the pupil — the subject has partially closed eyes — or to the subject not looking at the screen, typically when looking at the keyboard.

			Skills		
Subject	Time	Gaze.	Prog.	Scala	Rel. Alg.
1	0:48	86%	⋆⋆	⋆⋆	⋆
2	0:38	68%	⋆⋆⋆	⋆⋆⋆	⋆
3	1:08	61%	⋆⋆	⋆⋆	⋆⋆
5	0:51	78%	⋆	⋆	⋆⋆
7	0:48	71%	⋆⋆	⋆⋆	⋆⋆
9	0:47	76%	⋆	⋆	—
10	0:40	50%	⋆⋆⋆	⋆⋆	⋆
11	0:43	61%	⋆⋆⋆	⋆⋆⋆	⋆
12	1:07	62%	⋆⋆⋆	⋆⋆	—

Table 2. Various data about each subject: duration of the experiment and availability of gaze, as well as reported skill levels in programming, the Scala language, and relational algebra.

The number of subjects who took part in the experiment is relatively small, which may be an issue. Despite that, some of the results are statistically significant and even exhibit very strong effects. However, that increase the risk of unanticipated biases in the sample. The sample is representative with respect to reported programming and Scala skills, although the two are highly correlated. On the other hand, the sample does not contain subjects that are very experienced with the relational model. As was mentioned above, my subjective view is that the sample is fairly representative of students in computer science or engineering: it contains subjects with varying aptitudes to programming — both high and low achiever volunteered for the experiment — and at various points in their studies.

Figure 1 shows the proportion of time each subject looked at various elements on the screen. Time spent looking at code or at contracts represents 40–80% of the total. Time looking at the description of the relational model represents 5–10%. Compared to the length of the text, that is a relatively high percentage. The non-attributable time reported in this figure must be treated with caution: some of it, lost gazes, should be attributed to relevant categories. I cannot know how much of attributable time is in fact lost — as opposed to time spent looking elsewhere — and in which proportion to the various categories. Lost gazes are unlikely to introduce a statistical bias: it would be surprising if they were correlated with any other observed variable.

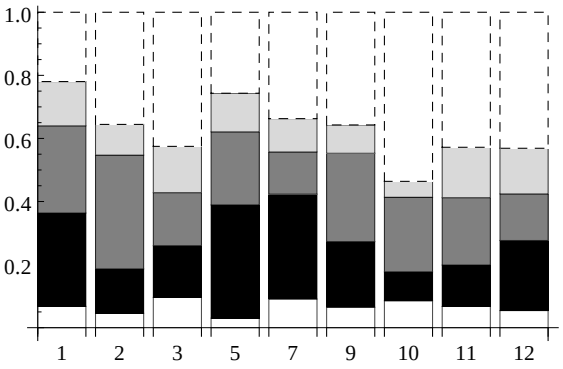

Fig. 1. Relative time for reading the relational model and the language documentation (white); reading code or contracts for algorithms 1, 2, and 3 (black, grey and light grey). The dashed part is non attributed time.

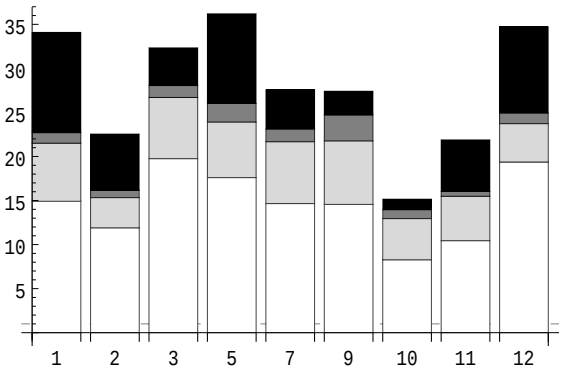

Fig. 2. Total time (minutes) for reading code tokens, by subject. Tokens are categorised into implementation (white), identifiers (light grey), and types (grey). The black part is the time used for writing contracts.

Figure 2 shows the time, in minutes, that each subject spent looking at families of tokens, and writing contracts. Times are quite variable and somewhat coherent with reported skill levels. It is interesting to notice that the time spent writing contracts varies widely. I cannot fully explain this variation, although anecdotal evidence obtained from replaying gaze movies makes me think that some subjects build a mental model of the code prior to writing the contract whilst other build the model whilst writing. Some subjects thus refined their contracts multiple times, in an increasingly conceptual style, whilst others wrote the final version immediately.

Results by coding style

Figure 3 compares the time spent looking at algorithms for each coding styles. I analysed the variance of reading times (using ANOVA) between the group "S/G" and the group "D/U and D/G". Reading time amongst these groups is different in a very significant way ($F_{1,25} = 34.7$, $p \leq 0.001$). On the other hand, there is no significant difference between the groups "D/U" and "D/G" ($F_{1,16} = 0.46$, $p = 0.51$). In other words, for the algorithms presented in this experiment, a coding style with a low cognitive density leads to an increased reading time, irrespective of the outcome. On the other hand, identifiers used as grounding hints have no measurable positive impact on that time, despite the fact that subjects had been reminded of the existence of the conceptual model that was used for grounding. However, the fact that the sample did not contain subjects with a thorough prior knowledge of the model must be taken into account.

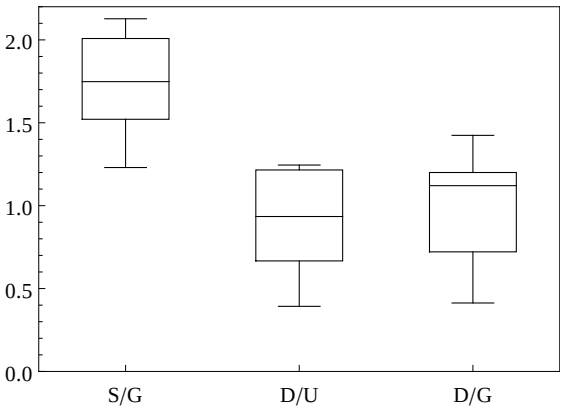

Fig. 3. For each style, the normalised times spent reading the code of algorithms.

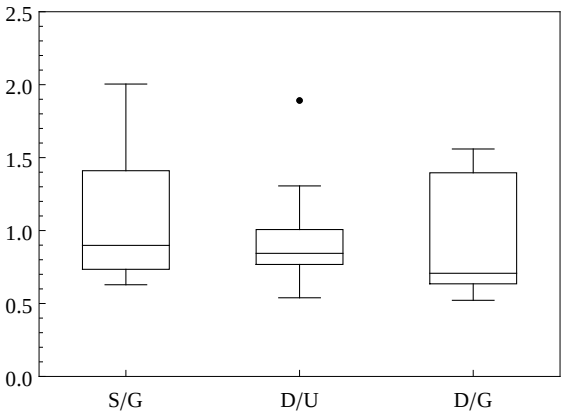

Fig. 4. For each style, the average normalised time spent looking at one token, independently of its nature.

The S/G style leads to much longer codes, as shown in section 3. Figure 4 aims to analyse how code length relates to comprehension time. It compares the time spent looking at each token, on average, for each coding style. An analysis of variance did not show any significant difference between the styles. The interval for mean differences between styles at a confidence level of 95% is approximatively between -0.3 and 0.5 (using the same normalised units as the figure). In other words, the data suggests that the time spent comprehending one token does not change significantly based on its cognitive content.

The major difference between the S/G style and the D/U and D/G styles is the time spent looking at grounding hints. In D/U and D/G styles, it represents about 10–15% of overall time; in S/G style, it is more than 25%. I postulate that the task of grounding the S/G code is harder than for the other two. On the other hand, the difference of time spent looking at non-grounded code is relatively minor between styles. Types play a minor role in all three styles. Arguably, types play their most important role in large applications, or in the definition of programming

interfaces and data structures. As such, algorithms like those used in the experiment may not be very representative of the usefulness of types. Figure 5 represents the time spent looking, for each style, at each family of tokens, as well as the time spent writing contracts. In line with the results mentioned above, the total time looking at S/G-style tokens is longer.

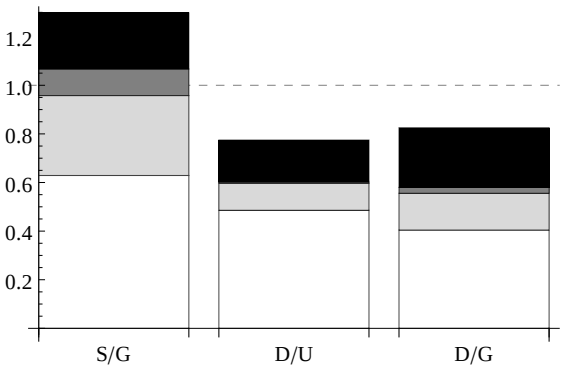

Fig. 5. For each style, the normalised time spent reading implementation tokens (white), identifier tokens (light grey), and type tokens (black).

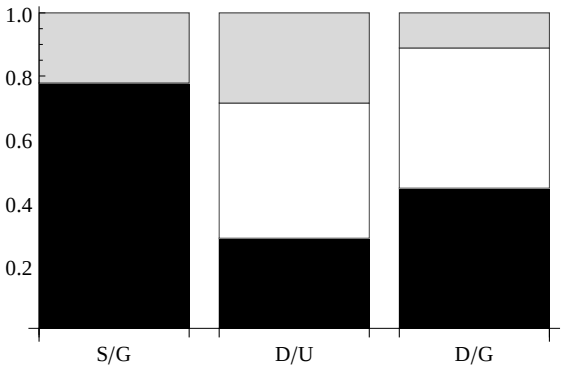

Fig. 6. Ratios of outcomes by styles. Black is a failed outcome, white is serviceable and light grey is conceptual.

Whilst the previous results are related to process variables, figure 6 charts the relation between coding style and the dependent variable of the experiment: the outcome of the comprehension task. As stated previously, the experiment was not actually designed to compare these variables, but one may be tempted to conclude, by looking at the graph, that D/U and D/G styles lead to increased code comprehension. The number of observations is too small to conduct Pearson's χ^2 test on all three categories of styles and of outcomes. After simplifying both categories — for style, D/U and D/G are merged into one category; for outcome, the serviceable and conceptual ones are merged — I could perform Pearson's χ^2 test in somewhat acceptable conditions. Its result ($X^2 = 3.74$, $p = 0.053$) is just too high to confirm a relation between style and outcome. On the other hand, many subjects informally told me that code written in the S/G style was "annoying" or "messy" and that they felt they had understood the code poorly. No remarks concerning the difference between the D/U and D/G styles were voiced. In fact, most subjects probably did not realise that there was a difference.

On Predicting Outcome

The measurement for outcome is too coarse to effectively serve as a statistical indicator of comprehension. Furthermore, a defect in the conception of the experiment means that self-reported confidence data cannot be exploited to refine the outcome data into something more relevant to the degree of comprehension. Despite that, intermediate variables are somewhat significantly predictive of the outcome. This is particularly true when differentiating between a serviceable and conceptual outcome.

Figure 7 and 8 show how outcome depends on intermediate variables. Reading time does not predict the outcome ($F_{2,22} = 1.72$, $p = 0.20$) and neither does it predict, assuming the result is successful, whether the outcome is serviceable or conceptual ($F_{1,10} = 1.78$, $p = 0.21$). The ratio of time spend reading identifiers — the primary form of grounding tokens — with respect to the overall reading time effectively predicts whether the outcome will be serviceable ($F_{1,23} = 11.8$, $p \leq 0.01$). The ratio of grounding tokens also predicts whether the outcome will be successful ($F_{2,22} = 5.86$, $p \leq 0.01$), but only because the variance of the serviceable outcome with respect to others is so large; this ratio cannot reliably differentiate between a failed and a conceptual outcome ($F_{1,16} = 0.37$, $p = 0.55$).

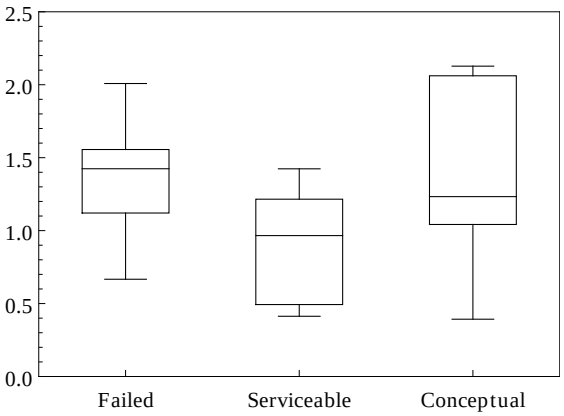

Fig. 7. For each outcome, the time spent reading algorithms.

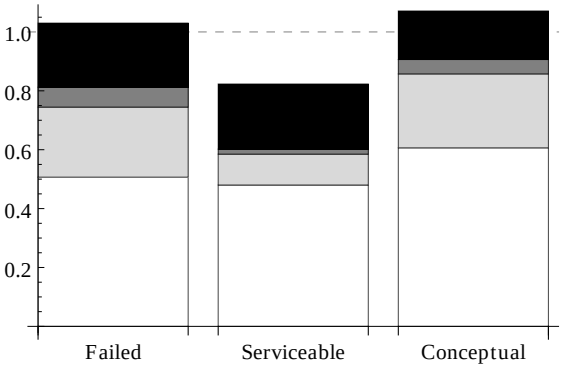

Fig. 8. For each outcome, the time spent reading implementation (white), identifiers (light grey) and types (black).

Fig. 9. A screen shot of the code as it was displayed during the experiment, with a "heat map" of relative gaze times. Left is algorithm 1 by subject 7 in S/G style. Right is algorithm 2 by subject 12 in D/U style. Note the fixations on the identifiers in S/G style with are absent in D/U style.

5 Discussion

Validity of the first hypothesis. The experimental results suggest that Scala code written using advanced constructs is superior to Scala code written in a style similar to that of Java. In terms of comprehension time, that difference is statistically significant despite the small sample size. In terms of the degree of comprehension that is achieved, there is anecdotal evidence — trends in the data and informal remarks made by subjects — that using advanced constructs makes the overall task of code comprehension simpler. It is interesting to note that the benefits of using Scala are visible even amongst a group containing programmers with a limited grasp of Scala's common grounds.

These results corroborate the first hypothesis, namely that code comprehension is improved when programmers can utilise common grounds to comprehend cognitively very dense code. I cannot say whether programmers only benefited from reusing their mental models for constructs, or whether the more literary style found in this code contributed to these results. The fact that subjects mentioned how "messy" and "annoying" the Java-like style was may however be interpreted in favour of the latter explanation. The fact that comprehension time per token did not differ based on the cognitive content of these tokens is unexpected. If this property can be generalised, it would give language designers a precise goal: the shorter the code, the better. It may also explain why domain-specific languages are so effective.

Validity of the second hypothesis. The experiment does neither strongly support nor strongly reject the hypothesis that grounding hints improve code comprehension. Subjects who had built a conceptual model of the code had spent more time reading grounding hints than those who had built a more technical model of the code. This result is difficult to analyse, but it may be used as an argument in favour of the second hypothesis. Many subjects spent a significant amount of time to write contracts of a relatively unsophisticated nature. The contracts had to be understood by a third-party (the experimenter). In line with the second hypothesis, the subjects would have to write contracts in a way that is meaningful within the common grounds they expect to share with the experimenter. The surprising amount of time that most subjects spent writing contracts shows that to be difficult. Overall, the experiment proved not very suitable for settling the second hypothesis' validity. Grounding code within a common-ground domain such as relational algebra may arguably become important only when considering large, complex programs. Still, it seems that a subject with a good grasp of relational algebra would have found it easier to comprehend the algorithms. On the other hand, the algorithms proved to be simple enough to be analysed without much reference to their model. Also, the sample was not completely suitable to analyse grounding, because most subjects lacked experience in relational algebra.

The design of the experiment made the data somewhat difficult to analyse. It would be useful, as future work, to run a similar experiment with a larger sample, truly independent measures, and using a grounding model that is better understood. It would also be interesting to test whether the difference measured between "styles" of Scala code could also be measured when different languages are used.

In conclusion, I believe that the results of the experiment do not contradict the distributed cognition model of software projects proposed in this article. On the contrary: the predictions of the model about the role played by common grounds in code comprehension are somewhat corroborated by the experiment. It remains an open question if other forms of knowledge, and other means of utilising common grounds for code comprehension exist. In reference to the title of this article, the experiment probably entitles me to conclude that programming languages are, in fact, improving as a medium for human communication.

Acknowledgment

I wish to acknowledge the contributions of Pierre Dillenbourg, Patrick Jermann, Phil Bagwell, and Martin Odersky.

References

1. Roman Bednarik, Niko Myller, Erkki Sutinen, and Markku Tukiainen. Program visualization: Comparing eye-tracking patterns with comprehension summaries and performance. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, pages 66–82, 2006.
2. Roman Bednarik and Markku Tukiainen. Analysing and interpreting quantitative eye-tracking data in studies of programming- phases of debugging with multiple representations. In *Proceedings of the 19th Annual Psychology of Programming Workshop*, pages 158–172, 2007.
3. Fred Brooks. *The Mythical Man-Month*. Addison-Wesley, 1995.
4. W. Douglas Brooks. Software technology payoff: Some statistical evidence. *Journal of Systems and Software*, 2(1):3–9, 1981.
5. Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. Object-oriented program comprehension: effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156, June 2002.
6. Thomas A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.
7. Cynthia L. Corritore and Susan Wiedenbeck. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human Computer Studies*, 50(1):61–83, January 1999.
8. Nick V. Flor and Edwin L. Hutchins. Analyzing distributed cognition in software teams: a case study of team programming during perfective software maintenance. In *Empirical Studies of Programmers: Fourth Workshop*, pages 36–64, 1991.
9. Edwin Hutchins. How a cockpit remembers its speeds. *Cognitive Science*, 19(3):265–288, July 1995.
10. Martin Odersky. *The Scala Languages Specification*. École Polytechnique Fédérale de Lausanne, January 2009.
11. Yvonne Rogers and Judi Ellis. Distributed cognition: an alternative framework for analysing and explaining collaborative working. *Journal of Information Technology*, 9(2):119–128, June 1994.
12. Helen Sharp and Hugh Robinson. A distributed cognition account of mature XP teams. In *Extreme Programming and Agile Processes in Software Engineering. 7th International Conference*, volume 4044 of *Lecture Notes in Computer Science*, pages 1–10, 2006.
13. Andrew Walenstein. Observing and measuring cognitive support: steps toward systematic tool evaluation and engineering. In *Proceedings of the 11th International Workshop on Program Comprehension*, pages 185–194, 2003.
14. Peter Wright, Bob Fields, and Michael Harrison. Analyzing human-computer interaction as distributed cognition: The resources model. *Human-Computer Interaction*, 15(1):1–41, 2000.
15. Iyad Zayour and Timothy C. Lethbridge. Adoption of reverse engineering tools: a cognitive perspective and methodology. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 245–255, 2001.