Biuletyn WAT
Vol. LXVIII, Nr 2, 2019

Code comprehension as a distributed construction of meanings

PIOTR COFTA

UTP University of Science and Technology, Faculty of Telecommunications,
Computer Science and Electrical Engineering, 7 S. Kaliskiego Str.,
85-796 Bydgoszcz, Poland, piotr.cofta@utp.edu.pl

Abstract: Code comprehension, a sub-domain of reverse engineering and software maintenance, does not provide useful explanation of common situations where developers, distributed and isolated from each other in time and space, come to a similar understanding of a code. This limits our ability to develop tools to support this popular aspect of code comprehension. This paper investigates this phenomenon from the perspective of sociology, intentionally distancing itself from the dominating psychological approach. The analysis, conducted mostly from the standpoint of social systems theory, highlights that as the construction of meanings is subjective, in the absence of any significant interactions, the dominant influence on the construction of meanings comes from current states of various social systems to which the developer belongs to. Thus, the similarity of meanings (hence a better comprehension of the code) can be achieved by understanding systems to which the developer belongs to and by coordinating their states.
Keywords: computer science, code comprehension, social systems theory, reverse engineering, software maintenance
DOI: 10.5604/01.3001.0013.3001

1. Introduction

It seems that research in code comprehension does not provide an insight on its key phenomenon: how is it possible that two people, at different time points and at different geography, come to the reasonably similar conclusion about the meaning of the piece of code? This paper intends to investigate this question from the perspective of sociology, thus distancing itself from the prevalent psychological view

on code comprehension. The outcome should inform the development of tools and methods used in code comprehension.

The paper attempts to shift the focus of code comprehension from psychology towards sociology, more exactly to social psychology and psycho-sociology. Unfortunately, the author was unable to identify any research from either domain that directly address the question of code comprehension. While this problem can be approached from either side, this paper focuses on introducing some social aspects to the process that is otherwise considered rooted in psychology.

There is a discrepancy between the number and the focus of psychology-oriented and sociology-oriented papers when it comes to software reverse engineering. Psychology is more prevalent, with Google Scholar (accessed 23/12/2017) reporting 13,700 references to psychology, versus 5,300 to sociology. Further, even the cursory analysis show that while psychological theories often form the foundation of those research, sociological theories are often used only to scope the problem.

It is possibly because as much as software engineering visibly became a social activity, reverse engineering is still considered an individual endeavour, best served by psychology. There is is a visible trend of collaboration in software engineering (but not in reverse engineering), with the wealth of research, tools and methods to support it, such as agile development or open source movement, but this trend did not found its way into the reverse engineering.

It is also possible that the obstacle is in the absence of the accepted theory of cooperation that can be applied to reverse engineering in the same way that theories of a collaborative work are applicable to software engineering. This also applies to code comprehension, being the discipline that is often considered a sub-set of reverse engineering [24]. This paper attempts to address this discrepancy.

The primary objective of this paper is to present the theory that can be used to construct a sociological approach to software reverse engineering, and specifically to code comprehension. That is the paper proposes the re-use of a particular social theory that can explain its perceived convergence of meanings, i.e. how it is that developers and maintainers, often separated in time and in space, come to the similar interpretation (and re-interpretation) of the code.

The proposition brought by the theory is that the meaning of the code is inherently subjective and it is constructed by each developer or maintainer separately, according to the state of all social systems they are members of. In the absence of any forms of actual interaction, the code can be comprehended well only if there is a similarity of states of those systems.

Casually, the quality of code comprehension is contingent on what can be called shared background and the similarity of experience of developers, neither of which is currently captured into knowledge databases or repositories.

2. Motivating cases

Code comprehension, while usually associated with reverse engineering and code maintenance, is a routine element of several software development activities. Following three cases are taken from author's experience with various development teams.

All those cases illustrate the question of how it is that the meaning of the code, comprehended by developers, is not fixed, is subjective, yet can be reconciled to some extent at another point in time and by another developer.

1. A developer wrote a code, and had it accepted (so at least it was a code that satisfied corporate standards). After few months, the developer looked at his own code again and honestly stated that he does not understand what it does. This — rather popular — case demonstrates that the comprehension of one's own code is not fixed in time. As memories of the process of writing the code deteriorate, so is the ability to comprehend it outright.
2. A developer wrote a piece of code in a rather haphazard style, not caring (and not knowing) much about any design patterns. Another developer looked at the same piece of code and immediately identified several design patterns that — to her — were apparent in the code.
   Here, the second developer discovers meanings that were not deliberately intended to be present in the code. It is not certain whether the first developer was a careless genius or whether the second one was trained to see patterns everywhere, but it is apparent that that meaning that the first developer intended to include in the code differs from what the second developer discovered.
3. The programming language used by the team introduced a raft of new, attractive features. Developers quickly embraced the new style that those features availed, but maintainers were reluctant. Eventually, maintainers could not easily comprehend newer pieces of the code. The tension was resolved by upskilling maintainers.
   This case demonstrates how the discontinuity in learning can have a negative impact on maintainers' ability to comprehend the code, affecting a relatively harmonious cooperation of both group. It also demonstrates that the correct solution is in assuring that both groups have a similar educational background.

3. Literature review

This literature review focuses on highlighting research that touched on social components of code comprehension, such as the use of domain knowledge, educational background, cooperation etc. The selection used Web of Science as a main search tool, using keywords such as "code comprehension" and "social".

Code comprehension is an important activity for developers and maintainers. During maintenance works they can spend 50% of their time and budget comprehending other's code [2], working mostly alone [9].

The exact definition of what constitute code (program, software) comprehension varies. Deimel and Naveda [7] define program comprehension as "the process of taking computer source code and understanding it". Rugaber [28] defines it as "the process of acquiring knowledge about a computer program" while Muller et al. [23] say that it is "the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself, to models of the underlying application domain, for maintenance, evolution, and reengineering purposes". The latter definition, as well as the review provided by Brooks [3], indicates that software professional will use the domain knowledge in the process.

The popular model of the code comprehension is provided by Letovsky [17]. The model stipulates that the 'assimilation' (i.e. comprehension) process of an individual that directs the mental model is driven by the available knowledge base and the external representation (the code). This model provides a clear link towards social sciences, as the knowledge base of an individual can potentially encompass all the experience (that comes from social activities) of such an individual.

Flor and Hutchins [10] focus on distributed cognition — the situation where developers jointly discover and comprehend the code, mostly through pair programming. The authors notice that this situation shifts the research from unobservable mental states to observable communication events, making them closer to the field of social sciences. A series of experiments demonstrates that distributed cognition has properties significantly different from the individual one.

Sim [31] provides a useful overview with a distinction between psychological and sociological studies of code comprehension, highlighting the philosophical difference between them. Sociological approach study people in a context of a group with research focusing on work practices and task performance. The overview provides an impression that the sociological approach is less mature than psychological studies.

Storey [33] lists several theories and tools that are applicable to code comprehension. However, almost all of them support individual comprehension, not a group one. Exceptions are for some general-purpose collaborative tools that can be also used for group comprehension, not the ones that are specific to group comprehension.

Ducasse et al. [8] propose an extensible re-engineering environment for tool integration with meta-model capabilities. The integration is achieved by introducing specific language-independent meta-model, that is based on versioned and annotated entities. The expectation is that such a tool will be able to convey the way the meaning has been constructed from one developer to another.

Meng et al. [21] consolidate variety of code comprehension tools using formal ontology-based models (meta-models) with a view to automate the process of integration and reasoning. While the approach focuses on supporting an individual

maintainer, there is a potential to use it to leverage knowledge acquisition and transfer within the group.

Damasevicius [6] stresses socio-technical nature of software development and indicates that design artefacts as well as relationships between them must be internalized by the team to become operational as mental models. Similarly, code comprehension should be considered a socio-technical activity.

Hamilton and Danicic [14] analyse dependence communities in large-scale code demonstrating that those communities decompose themselves into smaller ones along functional domains of the code, thus leading to sematic separation. It concludes that reverse engineering should be therefore best assigned to the group that is already familiar with the domain, not to a random maintainer.

Carneiro and Mendonca [4] discuss collaborative code comprehension, using the specific awareness model and they demonstrate that the collaborative environment to certain extent improves the ability to detect smells (i.e. deviations from good coding practices), despite collaborating groups being small.

Pereira dos Santos et al. [27], while discussing the use of social networks to support software development, indicate the importance of support for the maintenance phase, mostly in the form of better communication and collaboration tools.

Lungu et al. [19] stress the evolutionary and collaborative nature of code comprehension (focusing on architecture recovery). The process can be supported with the proposed tool that allows sharing and discovering the results of previous analysis sessions through a global repository of architectural views that allows for diverse views thus becoming an advisory tool for developers.

OMG [26] standardised a Knowledge Discovery Meta-model to represent existing software to offer interoperability and exchange of data between tools produced by different vendors, to retain, disseminate, and stabilise the knowledge.

4. Code comprehension and social activity

It has been already acknowledged that the social aspect of code comprehension is not prominently visible in research. It is not surprising, as code comprehension does not have typical characteristics of a social activity. As both the construction of meanings as well as the stabilisation of once constructed meanings require interaction [18], the properties of code comprehension listed below do not need themselves easily to support the construction of stable meanings.

- Solitary activity. Code comprehension is usually a solitary activity. Apart from occasional pair programming and collaborative code comprehension, the dominant industry practice is that code comprehension is a solitary activity of an individual who constructs the meaning of the code off the screen.

- Lack of feedback. Another feature of code comprehension is that the flow of information is only from developers to maintainers who want to comprehend the code, often with a significant time gap, often accentuated by the fact that original developers are no longer available. Thus, if there is any communication, it is devoid of any feedback: the maintainer neither can ask the original developer for a clarification nor he can engage in any meaningful discussion.
- Stored knowledge. The most efficient way to communicate the meaning of the code is through direct human interaction. However, those communication means are usually inaccessible to maintainers who must rely on stored information such as code and its metadata. This is because activities related to code comprehension are intermittent, with outbursts of activities separated in time. While stored information has some advantages when it comes to the retention of knowledge, it is seldom sufficient to reconstruct the original line of thoughts.
- Locality. The experience in comprehending one piece of code cannot be encapsulated into shared knowledge, the way readily available libraries encapsulate the developer's knowledge. Thus, neither the skill of code comprehension nor the comprehension alone scales up well.
- Imperfections. Code comprehension comes to play quite often when the code is not entirely correct, possibly often exactly because it is not correct. Consequently, code comprehension faces not a simple question 'what did the authors actually said', but a more convoluted one: 'what did the authors intended to say despite errors'.

5. The meaning of the code

Code comprehension deals with the construction of the meaning of the code, as referenced (usually in the passing, as a matter-of-fact) by several authors e.g. [25], [16], [35]. The view represented there is that, while software development deals with expressing business meanings in a form of a code, code comprehension is a reverse process of synthesising those meanings from the code. Both processes may have several stages, where code comprehension may e.g. establish meanings in a domain of software design, then architecture and only then in a domain of business logic.

Either way, code comprehension deals with meanings. 'Meaning' has itself several meanings, and this section serves as a clarification. Drawing from a tradition of semiotics [29], the 'meaning' of the code is what the code represents that is not in the representation itself, thus distinguishing between a sign (the code) and a signifier (the meaning). Within the scope of software reengineering, there is a clear distinction between activities that require only the cursory understanding the textual form of a code (such as re-formatting) and actions that require an understanding that goes beyond the code [36].

In the sociology, there are at least three approaches to the concept of 'meaning'. All of them can be also applied to code comprehension, as discussed throughout this section. These are: behavioural approach (that claims there is no meaning other than the one defined by the 'downstream' execution, reflecting the construct of semantics), hermeneutics approach (that claims that the only meaning resides 'upstream' with an original contributor) and communicative (that claims that meanings are subjective to contributors and that such subjective meanings may or may not converge).

5.1. Semantics – the behavioural approach

The behavioural approach states that there is no specific meaning attributable to the code other than the meaning defined by its execution. Thus, the code 'means' what the code 'does', as defined in terms of the platform that executes it. Hence, the meaning of any piece of the code is always fully explainable by the way it is executed. This is the approach adopted by the semantical analysis of software.

The name 'behavioural' used here is taken as an analogy to the behavioural approach (e.g. [32]), where the human being is denied any unexplainable mental life other than trained response to stimuli. In a similar manner, the code does not represent anything else but only its own functionality, as expressed by its execution.

When applied to code maintenance, this behavioural approach focuses on the ability to use and alter the code to fit the current purpose, without consideration whether such a purpose is compatible with the intended meaning of the code (as it assumes that the intended meaning of the code either does not exist or is irrelevant).

This approach lends itself to the analysis of the code 'as is', in a manner that benefits the hacking community (in both meanings of this word: those who want to attack the code and those who want to haphazardly make use of it). To certain extent, it also benefits those who e.g. would like to use the code without engaging into any deeper understanding, e.g. when they use the (potentially) unpublished API to achieve their objectives.

This is the area that can be heavily instrumented, but not easily abstracted. That is, there can be a variety of instruments that allow developers to experiment with the code and probe its execution such as debuggers, emulators, virtual environments, test cases etc. However, this approach is less likely to instrument means to abstract the code into higher level abstractions, as it presupposes that such abstractions do not exist.

The limitation of such an approach can be found e.g. in the combination of the reuse of the evolving code, the situation reasonably popular across the industry. Had the code been immutable, the pragmatic mastery of the code could have been sufficient and the code could have been reused as the developer pleases. However, without understanding what the meaning of the code is, it is easy to lose the ability to use such a code when it evolves.

Let's consider an example where there is an existing piece of code 'X' that executes the merge sorting algorithm, developed for some particular reasons. Such an algorithm is known to be stable and this is exactly what developer of the new piece of code 'Y' might be looking for. Therefore, the developer includes the call, and a dependency, from 'Y' to 'X'.

For as long as the 'X' remains unchanged, this situation benefits the developer. However, when the 'X' evolved, the new version may contain the sorting algorithm, but this time it could be the heapsort, which is the unstable one. After all, this may better satisfy 'X' and 'X' does not have any obligations to consider 'Y'. As the result, 'Y' is broken.

The fallacy of developers of 'Y' was to treat the code of 'X' 'as is', without understanding the meaning attributed to such a code by original developers of 'X'. Had they understood that 'X' requires any way of sorting, not this particular one, the problem would have not appeared.

5.2. Hermeneutics

Hermeneutics assumes that the original developer (the author) of the code had specific meaning of the code in mind (hence the code has some meaning attributed by the author). Therefore, to comprehend the code, it is necessary to understand (discover, reconstruct) this original meaning. The reader (the maintainer) must make all the reasonable effort to see the code the way the developer did.

Contrasting with the behavioural approach that bases on 'downstream' behaviour of the code, this one traces the 'upstream' behaviour — towards developers, design and business requirements. For example, certain control structures may reveal design paters used by the author, names of variable may provide clues regarding the expected function of those variables etc.

This approach lends itself to the study of a code to extract the meaning as it has been presented by the author. The field of semantics and semiotics, driven by hermeneutics (e.g. [11]) devoted an extensive amount of work into better understanding this approach, even though focusing on literary writings, not on software.

When it comes to code comprehension, the assumption of the existence of the meaning that can be extracted from the code is a salient feature of several threads of research. The subject of extracting original meaning from the text is extensively studied, both for the literary text and for the code, mostly from the psychological perspective. For example, characteristic elements of the code (known as 'beacons' or 'chunks' — see e.g. [20]) are often used by maintainers to guess the structure and the understanding of the code at the higher layer of abstraction.

It is interesting that incidentally such research and tools may attribute meanings where the meaning is not due. For example, had the author wrote the original code without any specific pattern, design or requirements in mind (say, quickly 'hacked'

it), the analysis of this code may still reveal some higher-level meaning that were not the intention of the author — as indicated by the second use case in this paper.

Hermeneutics approach would have rectified the situation described earlier in relation to the sorting algorithm: the developer of 'Y' would not rely on the code of 'X' without understanding what the authors of the 'X' wanted to achieve. Thus, the developer of 'Y' would be aware that the requirement of stability was not a concern of the original author, so that the call and the dependency should not be made.

However, it will fail to address what is the everyday experience in software development: that the code has many authors and contributors, possibly separated in time and space. Even if there was an original meaning attributed by the original author, this meaning evolved, possibly beyond recognition. In order to rectify such a problem, the communicative approach is needed.

5.3. Communicative approach

This paper focuses on the third approach: that the code has the plurality of its 'upstream' (more abstract) meanings. Specific meaning is subjective to the developer or maintainer, but those meanings may be reasonably similar [18], provided that there are favourable conditions for it to happen. This approach is in accord with various definitions and models of code comprehension presented in the literature review, in that they also indicate the possibility that different individuals may come to different comprehension.

The key point in the communicative approach is that despite such subjectivity it is possible to comprehend the code in a way that is sufficiently similar, thus enabling and facilitating maintenance works. This can happen without the need for an abstract 'true' meaning or for any forms of inter-subjectivity.

Thus, the key question, discussed in this paper, seems to be about conditions to make them similar, and the design of processes that support those conditions. The building blocks of this approach are: social systems, psychic systems (individuals), the construction of meanings and the special form of social systems – interactions. Those are discussed in more details in the next section.

6. The construction of meanings

The concept that the meaning of the code is maintained by the interaction represents the main proposition of this paper is the key proposition of this paper. This section will serve as a brief introduction to the social system theory (see Luhmann [18]), with focus on the creation of meanings and the role of interactions.

Note that the social systems theory, as its name suggests, does not deal with individuals, but with systems to which those individuals contribute. This may be

in contrast with everyday observations that the code comprehension is done by individuals. Still, the theory provides a vantage point that can be of value. The proposition, presented later in this paper, demonstrates how it can be applied to the case of individuals comprehending the code.

As Luhmann wrote in German, the term that he originally used for meaning is 'Sinn'. It is translated either as 'meaning' [18] or as 'sense' [22]. This paper uses the first interpretation, 'meaning', but in some places, refers to the second interpretation, 'sense', specifically when it refers to the system's activity of 'making sense'.

6.1. Introduction to social systems theory

This section provides a brief introduction to relevant elements of the social systems theory, using, whenever possible, references to the process of code comprehension. Out of necessity, it is both heavily abbreviated and simplified. For an authoritative reference see [18] and for an abbreviated version see e.g. [5].

The social systems theory is concerned with structures of autopoietic social systems, where each social system is a collection of communications between people. Even though people are not elements of those systems, systems require people to evolve. That is, the social system can be thought of as a structured, responsive collection of all relevant communications for as long as people who contribute to the system have ways to exchange those communications.

Note that people are not parts of social systems. People are however, 'irritated' by social systems (by new communications) and in response they 'irritate' social systems by adding communications.

The atomic component of a social system is the communication that is an inseparable fusion of information, as defined in [30], utterance (an external form of the communication) and understanding. The understanding is a distinction between the information and the utterance that defines the way the system responds to the communication with communications.

Code comprehension is an activity that apparently happens within the context of social systems, such as corporations, groups of people etc., but it also happens as an activity that seems to be solitary. However, no person stays outside all social systems, even though such person may not be even aware of them. While the fact of being a part of the software development team or of an organisation can be apparent, being an alumni of a particular college, a supporter of open software or an enthusiast of reactive programming also make a person part of various systems in a more obscure way.

6.2. Meaning

The 'meaning' is the way the system makes a selection how to respond to communications. In other words, the response of the system depends on the meaning that the system holds about the incoming communication. That is true for social systems, but it is also true for people who participate in those systems. The meaning that the system holds is the meaning that the person uses.

If one would like to describe the system in an algorithmic way, the closest reference would be the state-transition machine where the meaning represents the internal state of the system, and such a state decides how the system responds to the stimuli, while possibly changing the state at the same time. Eventually, the meaning becomes a successor of all previous communications and all previous meanings that the system held.

Individuals, in the process of contributing to the system, may construct their own meanings in a similar way, but those meanings are unobservable to the outside world, and always subjective. Individuals provisionally inherit the meaning from the system they belong to, reconstruct the meaning on behalf of the system and communicate the reconstruction back to the system to be incorporated into the system.

The existence of meanings is warranted by the complexity of the situation. The system must respond to every new communication in a reasonably timely manner, or it risks its own disintegration. Had the system been forced to consider all possible combinations of past communications only to find the way to respond to the current one, the sizeable system would have never been able to communicate at all. It is only because the system is selective in reactions by applying the synthesis of the existing communications, the response happens at all.

The meaning is not a point state, but rather a probability distribution across various states. Subsequent communications must always 'make sense', i.e. their understanding must always somehow refer to the existing meaning, even if by contradicting it. However, some of those communications may be closer to the centre of a meaning (i.e. make 'more sense') while others may be at the periphery of it.

There is no source of meaning external to the system – that is the meaning is always 'subjective' to the system. The meaning is no more than the convenient synthesis of all the communications that constitute the system. The meaning evolves as the system evolves, and every new communication alters the 'probability distribution' of the meaning that the system carries.

When it comes to code comprehension, the meaning that the developer attributes to the code is a consequence of meanings that are held by systems the developer belongs to. It is not only that "for a man with a hammer everything looks like a nail", but also that e.g. a person skilled in databases will see the existing code in a way that may be dissimilar to the view held by a person skilled in objective programming. They are, after all, shaped by systems they belong to.

6.3. Interactions

An interaction as a social system should not be confused with interaction as social activity, where communications are rapidly exchanged between parties. Interaction, as referred to throughout this paper, is a social system (so it consists of its own communications). Its specific purpose is to stabilise the meaning, and the interaction tend to have only one meaning to take care of. That is, the meaning (the sense) of an interaction is to stabilise the meaning that the interaction carry. The interaction is not concerned with the meaning as such (there is no preferred meaning), but rather with the cohesion of such meaning.

The interaction requires a very specific condition: a presence of psychic systems (people) who take active part in interaction. That is, while the social system is constructed outside of individuals, it still requires the active presence of individuals to stabilise the meaning — in essence to keep the social system working.

Again, risking over-simplification, if there is a meaning worth taking care of, there is an associated interaction that clarifies what the meaning is. Such a clarification keeps the 'probability distribution' reasonably compact by continuously exploring and challenging understandings that contribute to the meaning.

Code comprehension is not the activity rich in interactions between defined individuals. As already discussed, active presence of an author and a maintainer is not a popular way of working. However, systems that they belong to tend to exist much longer even though the interaction is possibly conducted by different individuals. Thus, it is possible that the carrier of the meaning of a code is not an individual but a system – or systems.

7. The proposition

The key question that this paper investigates is a simple one: how does it come that individuals, separate in time and space, come to reasonably similar conclusion about a meaning of a piece of code? And what can we do to improve their chances of those meanings to be similar enough to support the maintenance of the code?

The proposition is simple: it is because they are not acting as individuals, but as participants in social systems that shaped the way they look at the code and that will retain the meaning that they attributed to the code. In other words, it is the group of systems that maintains the meaning of the code, not an individual.

The following mode of social aspects of the code comprehension is proposed.

- Social systems retain meanings of how to interpret the code, i.e. how to attribute meanings to the code. As systems evolve, those meanings also evolves.
- Maintainer who attempts to comprehend the code is a member of several social systems, so the maintainer inherits from those systems several ways of interpreting the code. However, the maintainer may not use the most

recent versions of those meanings, depending on the intensity of his interactions within those systems.

- Once the code is comprehended, the outcome of such comprehension becomes the part of systems the maintainer belongs to. The speed and intensity of this process depends on the intensity of interactions the maintainer has within the system.

This paper particularly proposes to focus research on code comprehension on what happens before an individual encounters a code. That is, instead of positing the question of how the individual in a given settings approaches the process of comprehension (in a way characteristic to psychological sciences), it suggests to focus on how the individual has been formed by social systems up to this point – what meanings those systems imposed upon him.

7.1. An illustration

The model that emerges can be graphically represented as having several parallel lines, each representing changes in time to a meaning that the individual constructs from communications that he is engaged with. That is, it represents the development of a psychic system of an individual. Once the individual encounters a code, the meaning of the code is constructed depending on the current state.

Let's consider an example provided on Fig. 1. There are three persons there: the developer, internal maintainer, and external maintainer. The first two share the membership of the organisation system B, the latter one works for the organisation system C. Still, they share the common professional background of the function system A.

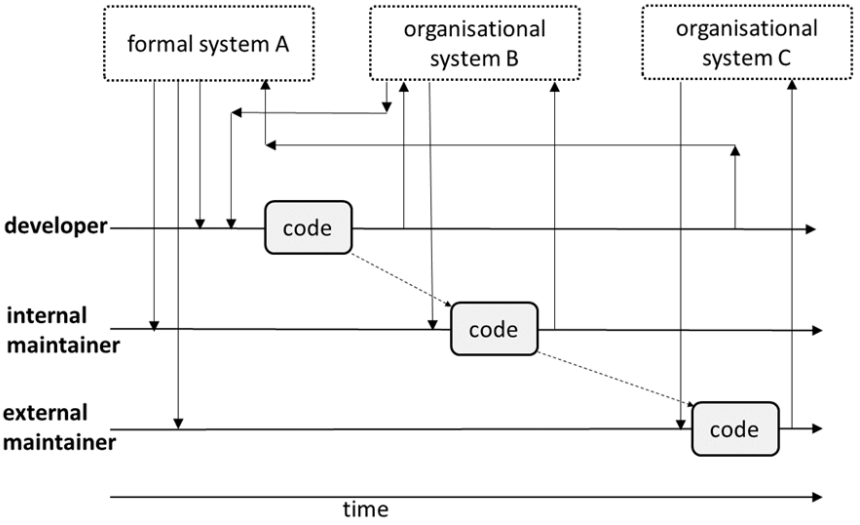

Fig. 1. An illustration of the example

The developer's last encounter with A has been some time ago, but he interacts within B quite often. As he encounters the code, his comprehension of the code is driven by some older meanings inherited from A and newer inherited from B. The meaning of the code, as discovered by him, is fed back into B (e.g. through local interactions), almost immediately, but it is fed to A only after a long time (e.g. at the conference).

When the internal maintainer experiences the code, his starting point is different. Meanings inherited from A are even older than in the case of a developer, but meanings inherited from B are not only fresh, but also already incorporate contributions from the developer. In a way he is in a privileged situation, assuming that B took care to protect the meaning generated by the developer through some forms of internal interaction (e.g. training). Even if B may not be able to interact directly with the developer, the influence may be strong enough for the internal maintainer to comprehend the code in a way that is very similar to the interpretation of a developer.

The external maintainer is in a relatively worse position, as he can rely only on the impact the developer made on A. certainly he is also affected by C, but C does not contain any meaning that can directly support the comprehension of the code. As the external maintainer encounters the same piece of code, his comprehension may be dissimilar and may lead to confusions. Further, as his comprehension will be fed back to C, any future encounter of the code by anyone from C may be influenced by his initial comprehension.

7.2. Observations

The naive interpretation of this model may lead to the conclusion that the identity of communications experienced by the individual before the exercise in code comprehension guarantees the identity of the meaning of the code. That is, if two individuals are subject to exactly the same stream of prior communications, they should attribute the same meaning to the code.

Luhmann [18] leaves no doubt that this is not the case, for at least three reasons. First, the individual acts for the benefit of the system, which means that the meaning of the code is contingent on the state of the system, not on the state of the individual.

Second, double contingency introduces instability that accounts for a quasi-randomness in the construction of meanings, so that even identical communications may lead to different reactions from individuals, thus diverging the paths that lead to the identity of meanings. It is only the interaction that maintains the similarity.

Finally, second-order observations means that the system can generate the meaning by self-analysing what it is already constructed of (i.e. past communications). Therefore, even without the inflow of communications, the system can alter its state in time.

Luhmann leaves also no doubt that there is no separate inter-subjectivity [13], as there is no separate place where it can exist and there is no separate system to maintain it. Systems can coordinate meanings only through interactions. However, as the construction of the meaning happens in time, divergence is neither immediate nor irreconcilable. If two individuals are left for too long to their own devices, the meanings will diverge. But if they reconciled the meaning quite recently, then for some time it will become similar.

This paper consequently proposes that in order to further code comprehension, we should consider what happened before an individual encountered the code, in relation to other individuals who encountered the same code as well as in relation to previous encounters within the same systems.

Specifically, we should consider forms of interaction that happen within the scope of systems, even if such interactions are not apparently related to the code. While the interaction always require presence, it does not mean that developers and maintainers have to meet in person. They can be engaged in some larger forms of interactions that do enough to stabilise meanings.

The paper expects that the similarity in the meaning of the code (that is, a successful code comprehension) is the outcome of a variety of social activities that shape systems as well as the individual such as education, meetings e.g. conferences, professional books, peer reviews and other forms of sharing, job mobility, membership in professional organisations etc. While none of those activities may directly aim at the particular piece of code, they all interactions that engage persons involved with the code.

8. Implications

There are several direct implications of the proposition when it comes to supporting the code comprehension. In general, they require to shift the interest, at least partially, from the psychological context of maintainers reading the code to the wider social context of developers and maintainers participating in the variety of interactions within and outside the context of the organisation.

It is, for example, possible that there is a correlation between the extent both developers and maintainers were involved in similar social activities (e.g. attended schools with similar curriculum, met likely-minded mentors or even red same books) and the quality with which they comprehend each other's code. Thus, the improvements in code comprehension may lie in creating such shared acculturation, e.g. in a form of standardised education or the growth in learning societies or even in a form of popular participation in the open code movement.

Further, this has implications on data that has to be captured to aid future comprehension. It may be that the important factor is to record the standardised

professional background of the developer or the maintainer, or to capture their contribution to other projects (in anticipation that the more projects they share, the more they are involved in the interaction).

As time progress, the 'old' meaning and the 'new' one may diverge. However, it does not mean that the new meaning as comprehended by the developer and the new meaning as comprehended by the maintainer must diverge. Assuming that they are still within the same social systems (e.g. by participating in the same conferences), their comprehension of the code may develop (and evolve) in parallel.

Further, there is an interesting question of analysing how to measure the discrepancy or the similarity of code comprehension and how to measure the similarity of experience of individuals involved. It may be that particular mental traits develop so early that only the standardisation of primary education will help, but it may be equally possible that even fully developed individuals can converge their meanings of allowed to interact.

Next, there is a question of how to construct the process by which developers can acquire experience that governs the creation of meanings. It is possible that this is a task for the educational forces, but it is also possible that the best approach would be to equip developers with tools that will allow them to capture and reflect on their approach to code comprehension.

This leads to the question of automation — to what extent code comprehension can be standardised through automation, and what is the best way to capture the collective expertise of developers. It is possible that solutions that allow developers to directly express their approach to comprehension may be beneficial.

9. Conclusions

This paper proposes to conceptualise code comprehension as a construction (and re-construction) of meanings, in expectation that this approach will address social aspects of comprehension, that are believed to be underdeveloped. The paper identifies some areas of deficiency, describes social aspects of code comprehension and then introduces concepts derived from the social systems theory that drives the conceptualisation, demonstrating its relevance to code comprehension. What follows is the review of areas that may address those deficiencies.

The proposed approach can bring several benefits.

- Re-focusing efforts towards social aspects of code comprehension. With the clear deficiency of research work on social aspects of code comprehension (as comparing with psychological aspects), the proposed approach introduces a solid theoretical platform that can both guide and integrate research in this area.

- Align with current trends in software engineering practices. Software maintenance and code comprehension is increasingly the collaborative work, and this trend should be reflected in research that can take into account the specificity of such a collaborative-yet-distant work.
- Anticipate growth in code re-development and re-use. The shared understanding of the code, fostered by well-maintained interaction may lead to further improvements in the re-use of existing code, possibly contributing to improvements in its quality.

The work has been funded as a part of the statutory research of the UTP University of Science and Technology, Bydgoszcz.

The article was prepared on the basis of a paper presented at the International Conference of Software Engineering KKIO'18. Pułtusk, September 27-28, 2018.

Received August 2, 2018. Revised April 4, 2019.

Piotr Cofta https://orcid.org/0000-0002-4269-6590.

REFERENCES

[1] Baker B., Gupta O., Naik N., Raskar R., *Designing Neural Network Architectures using Reinforcement Learning*, Published as a conference paper at ICLR 2017, https://openreview.net/pdf?id=S1c2cvqee, [accessed 11 December 2017].

[2] Boehm B.W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NH, 1981.

[3] Brooks R., *Towards a theory of the comprehension of computer programs*, International Journal of Human-Computer Studies, 18, 6, June 1983, 543-554.

[4] Carneiro G., Mendonca M.G., *SourceMiner – A Multi-perspective Software Visualization Environment*, Enterprise Information Systems: 15h International Conference, ICEIS 2013, Angers, France, July 4-7, 2013, Revised Selected Papers.

[5] Cofta P., *The Foundations of a Trustworthy Web*, Now Publishers Inc. Foundations and Trends in Web Science: vol. 3, no. 3-4, 2013, pp. 137-385. http://dx.doi.org/10.1561/1800000020.

[6] Damasevicius R., *Analysis of Software Design Artifacts for Socio-Technical Aspects*, INFOCOMP, v. 6, n. 4, Dec. 2007, p. 7-16, http://www.dcc.ufla.br/infocomp/index.php/INFOCOMP/article/view/190, [accessed: 3 February 2018].

[7] Deimel L., Naveda J., *Reading Computer Programs: Instructor's Guide and Exercises*, Technical Report CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie Mellon University, 1990, https://www.sei.cmu.edu/reports/90em003.pdf., [accessed: 15 December 2017].

[8] Ducasse S., Girba T., Lanza M., Demeyer S., *Moose: a Collaborative and Extensible Reengineering Environment*, 2005, http:// scg.unibe.ch/ archive/ papers/ Duca05aMooseBookChapter.pdf, [accessed: 3 February 2018].

[9] Fjeldstad R.K., Hamlen W.T., *Application Program Maintenance Study: Report to Our Respondents*, Proceedings GUIDE 48, Philadelphia, PA, April 1983, https://doi.org/10.1145/336512.336534.

[10] Flor N.V., Hutchins E., *Analyzing Distributed Cognition in Software Teams: a Case Study of Collaborative Programming During Adaptive Software Maintenance*, [in:] Empirical Studies of Programmers: Fourth Workshop, 1992, (eds.) J. Koenemann-Belliveau, T. Moher and S. Robertson, Norwood, NJ: Ablex.

[11] Gadamer H.G., *Truth and Method*, Continuum, London, New York, 2004.

[12] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object--Oriented Software*, Addison-Wesley, Boston, 1994.

[13] Habermas J., *The Theory of Communicative Action: Reason and the Rationalization of Society*, Polity Press, 1986.

[14] Hamilton J., Danicic S., *Dependence communities in source code. Software Maintenance (ICSM)*, 2012, 28th IEEE International Conference on, https://doi.org/10.1109/ICSM.2012.6405325.

[15] Jin D., Cordy J.R., Dean T.R., *Transparent Reverse Engineering Tool Integration Using a Conceptual Transaction Adapter*, Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003), Benevento, Italy, March 2003, pp. 399-408.

[16] LaToza T.D., Garlan D., Herbsleb J.D., Myers B.A., *Program Comprehension as Fact Finding*, Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE2007), Dubrovnik, Croatia, September 3-7, 2007.

[17] Letovsky S., *Cognitive processes in program comprehension*, Proc. First Workshop Empirical Studies of Programmers, Ablex Publishing, Norwood, N.J., 1986, pp. 58-79.

[18] Luhmann N., *Social Systems*, Stanford University Press, 1996.

[19] Lungu M., Lanza M., Nierstrasz O., *Evolutionary and collaborative software architecture recovery with Softwarenaut*, Science of Computer Programming, 79, 2014, 204-223, https://doi.org/10.1016/j.scico.2012.04.007.

[20] Mayrhauser A., Vans A.M., *Program Comprehension During Software Maintenance and Evolution*, Computer, vol. 28, issue: 8, Aug. 1995, https://doi.org/10.1109/2.402076.

[21] Meng W., Rilling J., Zhang Y., Witte R., Charl P., *An Ontological Software Comprehension Process*, Model. Proc. of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering, 2006.

[22] Moeller H.-G., *Luhmann Explained: From Souls to Systems*, Open Court., 2006.

[23] Muller H.A., Wong K., Tilley S.R., *Understanding Software Systems Using Reverse Engineering Technology*, 1994, https:// www.researchgate.net/profile/ Hausi_Mueller/publication/ 221233487_Understanding_Software_Systems_ Using_Reverse_Engineering_Technology/ links/ 00b7d515b635cbf536000000/ Understanding-Software-Systems-Using-Reverse-Engineering-Technology.pdf., [accessed 02 February 2018].

[24] Nelson M.L., *A Survey of Reverse Engineering and Program Comprehension*, April 19, 1996, ODU CS 551 – Software Engineering Survey, https://arxiv.org/ftp/cs/papers/0503/0503068.pdf., [accessed 18 January 2018].

[25] O'Brien M.P., *Software Comprehension – A Review & Research Direction*, Technical Report UL-CSIS-03-3, 2003, http://xyuan.myweb.cs.uwindsor.ca/480/P2_Review03.pdf., [accessed 03 February 2018].

[26] OMG – Object Management Group (2016) Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM). http://www.omg.org/spec/KDM/1.4. [accessed 11 December 2017].

[27] Pereira dos Santos R., Esteves M.G.P., S. Freitas G., Moreira de Souza J., *Using Social Networks to Support Software Ecosystems Comprehension and Evolution*, Social Networking, 2014, 3, 108-118. http://dx.doi.org/10.4236/sn.2014.32014.

[28] Rugaber S., *Program Comprehension*, 1995, http:// citeseerx.ist.psu.edu/ viewdoc/ download?doi=10.1.1.51.1404, [accessed 28 January 2018].

[29] Saussure F., *Course in General Linguistics*, Bloomsbury Academic, 2013.

[30] Shannon C.E., Weaver W., *The Mathematical Theory of Communication*, University of Illinois Press, 1949.

[31] Sim S.E., *Supporting Multiple Program Comprehension Strategies During Software Maintenance*, Master's Thesis, Department of Computer Science, University of Toronto, 1998.

[32] Skinner B.F., *Beyond Freedom and Dignity*, Hackett Publishing Company, Inc., 2002.

[33] Storey M.A., *Theories, methods and tools in program comprehension: past, present and future*, Program Comprehension, IWPC 2005. Proceedings. 13th International Workshop on. https://doi.org/10.1109/WPC.2005.38.

[34] Taherkhani A., Malmi L., 2014, *Beacon- and Schema-Based Method for Recognizing Algorithms from Students' Source Code*, Journal of Educational Data Mining, vol. 5, no 2, 2013.

[35] Tiemens T., *Cognitive Models of Program Comprehension*, 1989, https://pdfs.semanticscholar.org/052e/4a854a35c5e92b9c91fddfdfe07da119f83b.pdf, [accessed: 2 February 2018].

[36] Tonella P., Potrich A., *Reverse Engineering of Object Oriented Code*, Springer, 2005.

P. COFTA

Podejście do zrozumienia kodu jako do konstrukcji sensu

Streszczenie: Rozumienie kodu, istotna część inżynierii oprogramowania, jest obecnie badane głównie z pozycji psychologii, a w znacznie mniejszym stopniu z pozycji socjologii. Przypuszczalnie spowodowane jest to odczuciem, że dostępne teorie socjologiczne nie odnoszą się do problemów związanych z rozumieniem kodu. Ten artykuł argumentuje, że socjologiczna teoria systemów społecznych może znaleźć zastosowanie w badaniach nad rozumieniem kodu. Proponuje on skoncentrowanie się na formach systemów społecznych, które pozwalają na spójną rekonstrukcję znaczenia kodu. Artykuł ilustruje rozważania serią przypadków użycia, demonstrując, że rozumienie kodu jest i powinno być traktowane jako działanie społeczne, opisywalne odpowiednimi teoriami. Następnie skoncentrowano się na proponowanym wykorzystaniu teorii systemów społecznych, aby zakończyć omówieniem potencjalnych implikacji nowego podejścia na różne obszary badań.

Słowa kluczowe: informatyka, rozumienie kodu, teoria systemów społecznych, inżynieria odwrotna, utrzymanie oprogramowania

DOI: 10.5604/01.3001.0013.3001