Towards Employing Informal Sketches and Diagrams in Software Development*

Milan Jančár[1] and Jaroslav Porubän[2]

1 **Department of Computers and Informatics, Technical University of Košice, Košice, Slovakia**
`milan.jancar@tuke.sk`
2 **Department of Computers and Informatics, Technical University of Košice, Košice, Slovakia**
`jaroslav.poruban@tuke.sk`

Abstract

Programmers write notes and draw informal sketches and diagrams. We hypothesize about understandability and helpfulness of these sketches and their deeper inclusion into software development process. We are leveraging the fact that we have a collection of such sketches affiliated to a commercial software system. We have the opportunity to study sketches that were created naturally, not intentionally for research purposes. The oldest sketch was created a year and a half ago and the most recent one a half a year ago. Our initial experiment shows that these sketches are pretty understandable even after some time – even for another person.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, D.2.7 [Distribution, Maintenance, and Enhancement] Documentation

Keywords and phrases sketches, diagrams, design, maintenance, comprehension

Digital Object Identifier 10.4230/OASIcs.SLATE.2017.4

1 Introduction

Many developers spontaneously write or create notes, lists, tables, ER/class diagrams, drawings etc. [4] For the sake of brevity, let us refer to all kinds of these informal artifacts simply as a *sketch*. There are various reasons why these sketches may come into being, mainly "to understand, to design and to communicate" [3]. Many important design decisions are made on whiteboards [3]. These sketches may resemble UML but do not strictly adhere to it [4], they are spontaneous, ad-hoc and informal. We believe this spontaneity and informality is great to capture *immediate* thoughts. These sketches can be later changed, refined, even formalized (if necessary) – usually such sketches also have a longer lifespan and are more likely to be archived [1]. An interesting fact is what medium developers use for sketching: almost two thirds are on some analog medium (mainly paper and whiteboards), whereas modern means such as *interactive* whiteboards, tablets and smartphones are almost never used [1].

It is not sufficient to just *archive* those sketches – some *organization* must be brought in. Not to forget about the time dimension – as mentioned, the sketches *evolve* and we need means for handling this. Baltes et al. [2] created a tool for managing these sketches, especially for linking them to relevant parts of source code. There is still room for improvement and

* This work was supported by the project KEGA No. 047TUKE-4/2016: "Integrating software processes into the teaching of programming".

for studying usability and influence of sketches on program comprehension when embedded to corresponding code fragment.

We believe the potential of these sketches is yet to be fully utilized and we have a direct experience with these sketches, which in fact inspired this kind of research. It must be emphasized that we are only in the early stage of the research and the paper presents preliminary results obtained by observing and analyzing the sketches of one programmer. The aim was to better understand this field of study. Future work will focus on more rigorous experiments involving more programmers. The ultimate objective of our future work is to validate a deeper inclusion of sketches into software development process and to propose new approaches and recommendations for developers and for creators of those sketches to better utilize their potential.

We collected sketches capturing internals of a commercially developed system we participate on. The added value of our initial experiments lies in the fact that those sketches are *quite old* – their origin is in range from *a year and a half ago* to *a half a year ago* (October 2015 – October 2016).

The contribution of the paper lies in the *analysis and evaluation of understandability and (perceived) helpfulness of sketches* after a relatively long time period since their origin. In summary, the following aspects are regarded:

- the *nature* of the sketches – type of elements, notations, etc.,
- perceived and objective *understandability*,
- perceived *helpfulness*,
- perception of the sketches by a *non-author*.

2 Background

A little context is given – an explanation of the origin of the sketches and a brief description of a software project they are related to.

2.1 Software System

The software system documented by the mentioned sketches has a following nature. The system is a company internal system for managing their customers (people applying for a job). It is a web application consisting of a server side created in Grails (a Groovy-based framework for the Java Virtual Machine) and a client side created in AngularJS (a JavaScript framework). Source files have ca. 150 KLOC[1] and following programming languages are used[2]: Groovy (48%), JavaScript (28%), HTML (14%), Java (7%) and some others.

2.2 Origin of Sketches

The sketches are private notes taken by one of the authors of the paper. It is important to note that they were *not* intended for such a research. Their studying is purely incidental. Knowing about the consecutive research beforehand might have invalidated obtained results because the sketches might have been (subconsciously) adjusted or a greater effort might have been put into creating them (cf. Hawthorne effect).

[1] Kilo Lines of Code (no blank or comment lines), based on the output of `cloc` tool applied to files stored in a Git repository (`git ls-files`).
[2] As reported by *GitLab* in Graphs–Languages section.

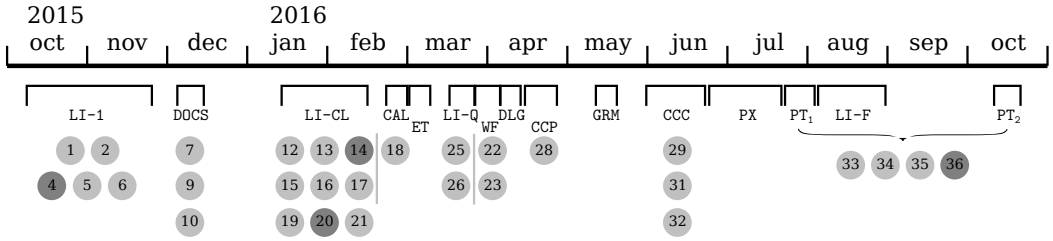

Figure 1 Time frame of sketches, the dark ones were selected for closer studying.

3 Data Collection

We are taking a collection of 37 sketches and are trying to get as much useful information as possible from it.

First, we assigned identifiers to *all* of the found sketches in the exact order as they have been found filed, starting with the paper on the bottom given the ID 1.

Then, from *37 sketches*, 7 were discarded (namely # 3, 8, 11, 24, 27, 30, and 37) because they were archived either by mistake or because they were remains of to-do lists (crossed out to large extent) with very little information value. So we have *30 sketches* for further analysis.

The next important thing, since we are focusing on the fact that the value of sketches lies in their *age*, was to define a time frame in which the sketches originated. That turned out nontrivial since the sketches were not timestamped. So we chose the following approach:

1. Utilizing a Git repository of the affiliated software project, we listed all commits by the author of the sketches and skimmed over them to get a list of tasks being done on the project plus their time range. By *a task* we mean some self-contained functionality, a feature addition/improvement, we might say *a user story* (although we do not want to imply any development methodology) which is sufficiently significant (lasting at least five days).
2. We analyzed sketches to assign a task from the list obtained in the step 1 to each sketch.
3. Based on the assignment *sketch→task* (step 2) and *task→time* (step 1) we can approximately define a time of origin of respective sketches. See Fig. 1 for the resultant time frame. (Tasks are assigned IDs such as `LI-1`, `DOCS`, but their meaning is internal and not important for this study.)

4 Nature of the Sketches

Technically, all but two sketches were created by hand on a piece of (scrap) paper, those two were drawn on a computer (by another person). The most of them (ca. 80%) were on a single page A4 paper.

We analyzed types of elements and notations used in sketches. When appropriate, well known standard notation elements were used such as those for entity/relation diagrams, class diagrams, state transition diagrams etc. However, they were not used precisely, rather freely with additional useful ad-hoc symbols.

A summary of observed recurring patterns found in sketches is in Table 1. Number of sketches having a particular type is stated. The sum does not yield 30 (100%) because one sketch may contain elements of more than just one type.

Table 1 Types of sketches.

Type	Abs. no.	Rel. no.
entity/relation (ER) diag.	9	30%
domain entities and relations between them		
algorithm	7	23%
important steps, strategies, if-then cases		
user interface sketch	5	17%
a graphical user interface (GUI) design		
state transition diag.	4	13%
a diagram of states and transitions between them		
table/matrix	4	13%
arbitrary information captured in a table/matrix manner		
Q/A list	4	13%
a list of questions and answers which are obtained from a more experienced fellow developer		
modules overview	2	7%
a high-level view on modules structure and their dependencies		
generic notes	18	60%
notes, prevalently in form of a list, not matching any above-mentioned characteristic		
all	*30*	*100%*

5 Initial Experiment

We ask these questions: *Are sketches, after a time has passed, understandable for their author and/or for other people?* And if the answers is yes: *Are those sketches still helpful, for instance, in maintenance tasks?*

From the author perspective, sketches are still very well understandable, almost all notation symbols and abbreviations are familiar.

Nonetheless, to evaluate a hypothetical understandability and (perceived) helpfulness of the personal sketches by someone else, we performed a quick exploratory experiment testing a rather vague hypothesis that *personal sketches related to a software project, even after some time, may be understandable and helpful for other developers participating on that project.*

We chose a questionnaire approach. A respondent is our colleague who also works on the mentioned software project. Thus he has some general domain knowledge. However, he has not worked on the tasks the sketches were about. He could have some marginal knowledge about them, though.

There are probably two main factors impairing understanding for him: *a time factor* (sketches may be outdated) and *"created by someone else"* factor. On the other hand, there is one thing that probably has a positive effect on understanding – the mentioned fact that the respondent has an experience with the related system.

5.1 Method

The questionnaire was comprised from the following 5 questions.

- **Q1:** Which project artifacts (on a file level) are related to a given sketch? To what extent – on the scale: marginally (less than 1/3 of a file content), partially (more than 1/3 of a file content but less than 2/3), largely (more than 2/3 of a file content)? [Time to solve: 10 minutes max. per a sketch]

- **Q2:** The given sketch involves various texts, pictograms, abbreviations, symbols, notations, diagram elements etc. Subjectively, to what extent do you think you understand the sketch? The scale: 0–20%, 20–40%, 40–60%, 60–80%, 80–100%.
- **Q3:** What hinders you the most in understanding the given sketch? (Open question)
- **Q4:** With the statement "The given sketch will help me in the future solve related maintenance tasks (adding or modifying a feature, bug fixing)", I: a) strongly agree, b) agree, c) do not know to take a stand (neutral), d) disagree, e) strongly disagree.
- **Q5:** Briefly summarize what you have grasped from the given sketch. If you caught some interesting details, include them. (Open question)

The 1st and 5th questions examine indirectly the respondent's *objective* level of *understanding* (when compared to the reality). The 2nd question targets the *perceived* level of *understanding* and the 4th question the *perceived* level of *helpfulness*.

For the sake of simplicity, only four sketches were selected for this initial experiment. They were selected in such a way that the following qualities were covered in the widest possible spectrum: age, neatness, notation types. Their ordering reflects our assumptions about their difficulty to be understood by a stranger (starting with the presumably easiest one). They have the following characteristics (for their age, see Fig. 1):

1. **ID 20:** a neat diagram, adhering (although not strictly) to well-known UML class diagram notation, it contains 7 classes (domain entities); we consider it easy to understand, it is basically a "control" sketch,
2. **ID 14:** basically a state transition diagram; contains some unexplained symbols and notations which makes it nontrivial to understand; a context (states and transitions of *what*) is not explicitly given,
3. **ID 36:** an intricate diagram drawn with no standard notation in mind; captures relationships among controllers and services with some additional notes; contains many unexplained abbreviations what makes it even harder to understand,
4. **ID 4:** has a form of a table; contains notes describing a strategy (algorithm); contains many unexplained abbreviations and also notations such as circles, arrows and wavy lines.

To create an image of what the sketches used in the questionnaire look like, photos have been taken, see Figures 2, 3 and 4.

5.2 Results

The answers to the questions Q2–Q4 from the filled out questionnaire are in Table 2. For the Q1, *we* decided which artifacts should be considered related, hence "correct", as we have a deep understanding of the system and also sketches. Based on this, we computed recall and precision of artifacts included in the obtained answer for this question.

A little elaboration is needed to clarify how we computed recall and precision. Let us have two sets: a set of "correct" (relevant) elements and a set of "selected" elements. It is well known that we compute recall r and precision p as follows:

$$r = \frac{|correct \cap selected|}{|correct|} \quad p = \frac{|correct \cap selected|}{|selected|}\,.$$

In our case, source artifacts (files) are those elements. However, our respondent was asked to not only select related artifacts but also to state an extent (marginal, partial, large) to which they were related. To regard extents of artifacts, we assigned weights (1–marginal, 2–partial or 3–large) to the artifacts in both sets (of selected and correct artifacts), effectively creating two (so-called) weighted sets or *multisets*. Above-mentioned formulas still hold but

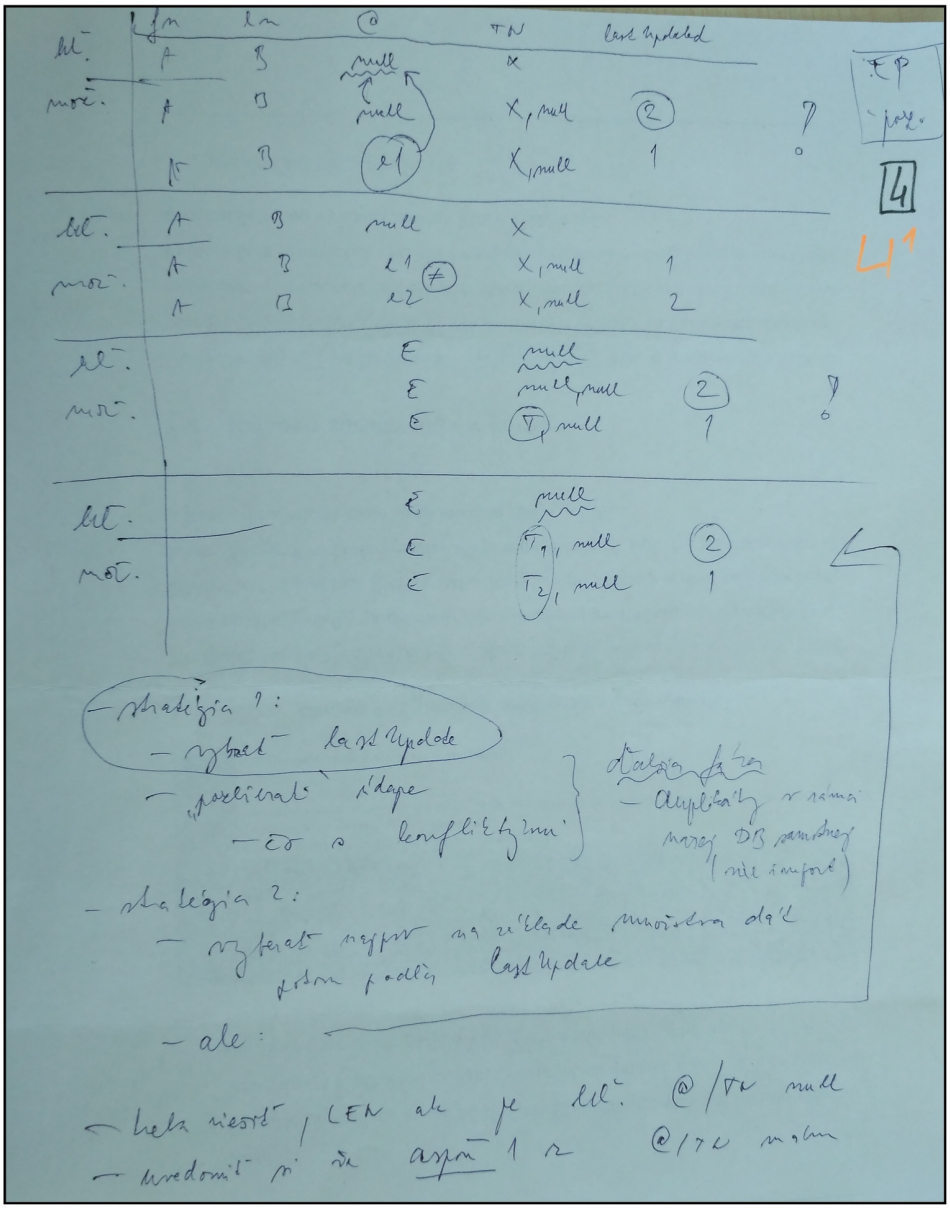

Figure 2 Sketch #4.

correct and *selected* are now multisets, and thus special rules, e.g. for intersection, apply. For instance, let *correct* multiset be $\{a, a, a, b\}$ (meaning an artifact a to a large extent and an artifact b to a marginal extent) and *selected* be $\{a, a, c\}$. Then, their intersection is $\{a, a\}$ – meaning "the artifact a to a partial extent".

Answers to the question Q5 and a high rate of confidence expressed in the Q2 aroused our interest. We extended the given textual questionnaire by interview questions asking about concrete facts captured in a sketch in order to find out if the respondent really understood them (or just *thought* he understood them).

Conclusions drawn from the Q5 combined with the interview are in Table 3.

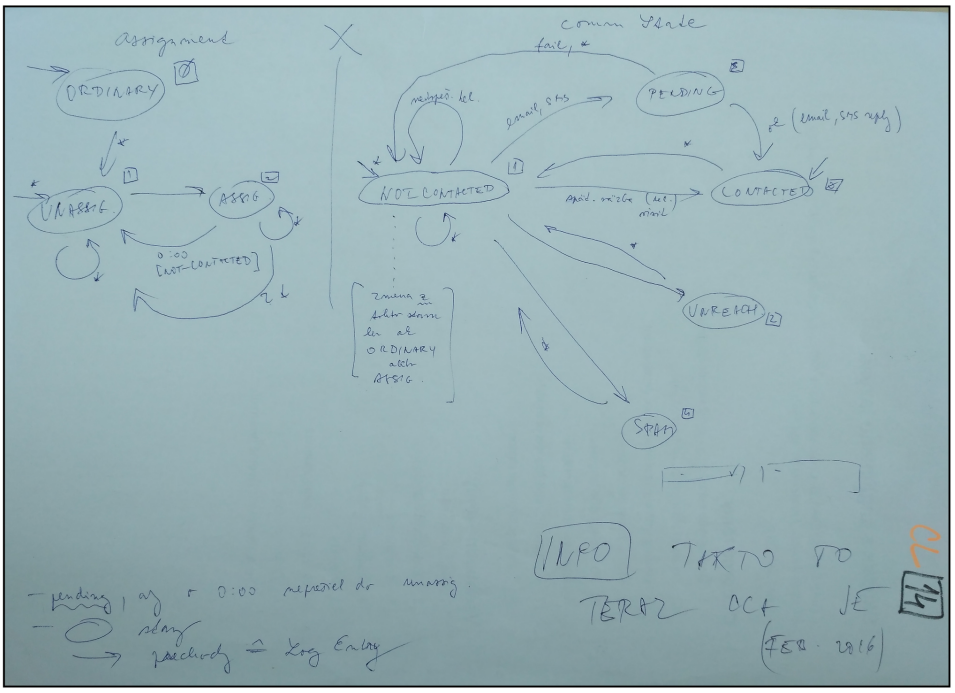

Figure 3 Sketch #14.

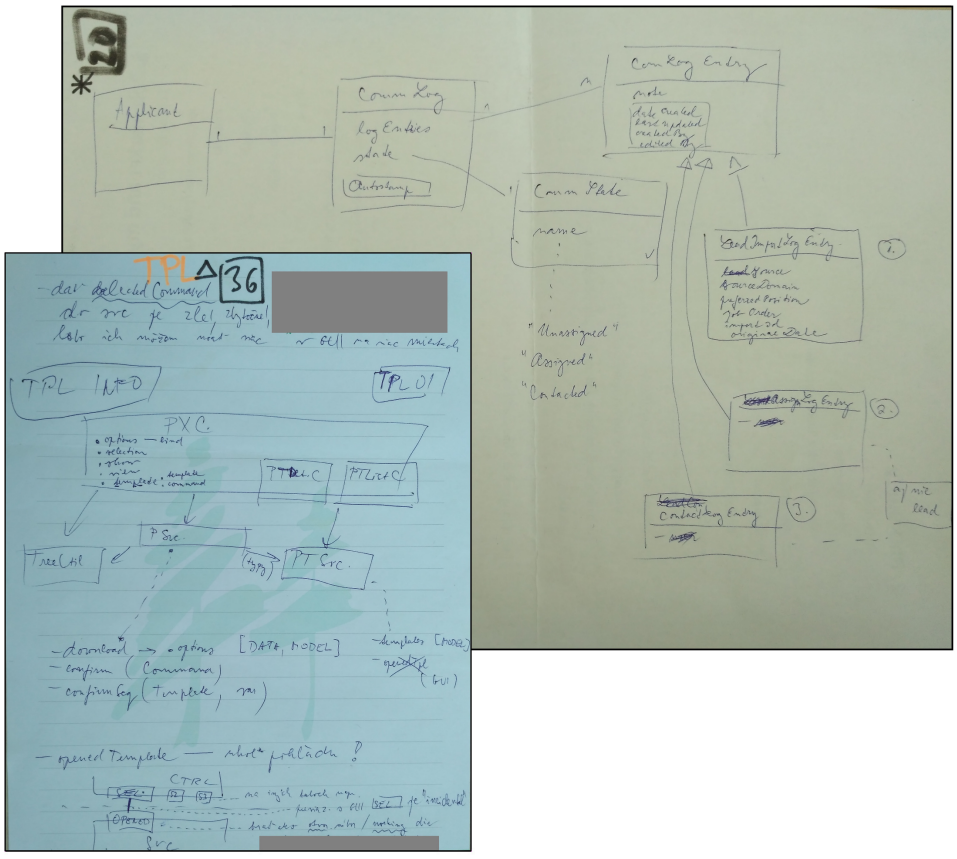

Figure 4 Sketches #20 and #36 (top-down, blank space overlapped).

Table 2 Obtained results from questionnaire.

Sketch	Q1 (artif.)	Q2 (perceiv. und.)	Q3 (hindrance)	Q4 (perceiv. help.)
#20	$r = 84.6\%$ $p = 64.7\%$	80–100%	illegible handwriting, notation (numbers)	agree
#14	$r = 37.5\%$ $p = 25.0\%$	60–80%	illegible handwriting, notation (a math sign)	neutral
#36	$r = 0.0\%$ $p = 0.0\%$	0–20%	illegible handwriting, abbrevs., small text, notation (rectangle, arrows)	disagree
#4	$r = 100.0\%$ $p = 66.7\%$	80–100%	illegible handwriting	agree

Table 3 Question Q5 and the interview.

Sketch	Report on the answer to Q5 and additional interview questions
#20	understood well, no more interview questions asked
#14	understood pretty well, although 1 fact was misunderstood and 2 symbols (*, ×) not understood
#36	completely not understood/misunderstood; misunderstood abbreviations and illegible handwriting caused that the respondent saw totally unrelated elements
#4	understood well, all additional questions (except for one) answered correctly or satisfactorily

5.3 Discussion

Evaluating the question Q1 turned out to be very difficult. There are significantly different views on what it means for an artifact to be related to a sketch. That caused relatively low values of recall and precision even for the sketch #20 where we thought the artifacts are obvious. Differences were also caused by various opinions on "covered extent" of artifact.

The perceived understanding (Q2) nicely correlate with the perceived helpfulness (Q4) so the respondent thinks that if he can understand a sketch, it will also help him.

A high level of confidence (Q2) was suspicious but based on Q5 and the additional interview questions, it seems that the respondent really understands what he claims. It is most surprising at the sketch #4, which we considered to be hardly understandable but the respondent understood it pretty well. The respondent admitted working on a task similar to the one described by this sketch. But still, he had to infer the meaning of obscure abbreviations and unexplained notation (arrows, circles, lines) and that was nontrivial.

What hinders understanding the most (Q3) is obvious and expected – mainly *illegible handwriting* and *unexplained notation*; also abbreviations, but not that much, probably because the respondent (a fellow developer) shares some domain knowledge and is able to infer them.

Sketch #36 came as a surprise: not because it was not understood but because the respondent was able to see *something what was not there*. It was caused mainly by ambiguous abbreviations and notation.

6 Future Work

Our future work may focus on measuring the *helpfulness* of embedding (or linking) sketches directly in (to) source code for *program comprehension*. "Program comprehension is an internal cognitive process that inherently eludes measurement" [5]. Measuring program comprehension (understanding) is hard and only *indirect*. Siegmund [5] enumerates the following approaches for measuring:

1. *software measures* measuring the code itself: based on the assumptions that the more complex code, the harder it is to comprehend – e.g. lines of code, cyclomatic complexity;
2. *subjective rating* of developer's understanding;
3. *performance of developers*: based on the assumed correlation between ease of understanding and speed of fulfilling a given task;
4. *think-aloud protocols*: allow observing a process of comprehending by verbalizing subject's thoughts.

Considering these standard approaches, software measures (1) are not applicable for obvious reasons: our approach does not affect existing code. In (2) and (4), measurements would be hardly comparable and obviously biased. The approach (3) – measuring performance of developers – seems like the most appropriate choice.

We plan to conduct a controlled experiment, where participants (programmers) will be given a task and source code of the relevant part of the project. A task may be oriented on programming (altering a functionality) or deriving some knowledge from the code. Both approaches require *comprehending* and can be measured – by measuring a time to fulfill the task. Participants of the experimental group will also be given a relevant *sketch*, since we are hypothesizing about their *helpfulness*.

7 Conclusion

Our small experiment showed that sketches, even after a long time, have a potential to be understandable and helpful – even for other people.

We will conclude this paper with lessons learned from our initial experiment about utilizing sketches in software development:

- One should care about his or her handwriting and to explain abbreviations and especially notations used.
- Being consistent in abbreviations and notations across many sketches increases the chances of their understandability.
- From various reasons, it might be useful to timestamp all sketches.
- It is also a good practice to label sketches with a related task – while a sketch is current, its context is obvious, but later the context is lost. Also these labels may streamline sorting out sketches or finding sketches related to a specific subject (task).
- We observed the following three reasons why a sketch was created: (1) to help think/reason, (2) as a medium to ask questions and capture answers (for instance from a fellow more experienced programmer), (3) to capture important design decision for future reference.

As the biggest hindrance to fully employ sketches in the development process, we consider the fact that those sketches are put away, archived, with no live connection to software source artifacts. They simply do not automatically pop up when we need them and this also is the area of our future interest.

References

1 Sebastian Baltes and Stephan Diehl. Sketches and diagrams in practice. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 530–541, 2014.

2 Sebastian Baltes, Peter Schmitz, and Stephan Diehl. Linking sketches and diagrams to source code artifacts. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 743–746, 2014.

3 Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *SIGCHI conference on Human factors in computing systems*, pages 557–566, 2007.

4 Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. How software designers interact with sketches at the whiteboard. *IEEE Transactions on Software Engineering*, 41(2):135–156, February 2015.

5 Janet Siegmund. Measuring program comprehension with fMRI. *Softwaretechnik-Trends*, 34(2), 2014.