An Observational Study of How Experienced Programmers Annotate Program Code

Craig J. Sutherland(✉), Andrew Luxton-Reilly, and Beryl Plimmer

University of Auckland, Auckland, New Zealand
{cj.sutherland,a.luxton-reilly,
b.plimmer}@auckland.ac.nz

Abstract. This study investigates how and why experienced programmers annotate program code. Research has shown that marking up prose with a pen is an invaluable aid to reading for understanding. However program code is very different from prose: there are no studies on how programmers annotate code while reading. We asked experienced programmers to read code printed on paper and observed their annotation practices. We found the main reasons for annotating code are to assist with navigation and to record information for later use. Furthermore, we found annotation practices that are hard to replicate in current standard Integrated Development Environments. This suggests that support for digital ink annotations in programming tools may be useful for comprehending program code.

Keywords: Freeform annotation · Reading code · Understanding code · Observational study

1 Introduction

Reading code for understanding is an important part of maintaining computer applications [1]. Programmers can spend a significant portion of their time trying to understand code [2]. But reading for comprehension is not limited to software development, people read for comprehension for many different tasks [3]. What is unique about reading program code is it is almost exclusively read on a computer screen.

Programmers will often use tools when trying to understand code [4]. These tools can assist by providing alternate ways of navigating and different visualizations of the code. They attempt to reduce the amount of work the programmer has to do so the programmer can focus on comprehension [4]. However research has found that programmers do not always use these tools. Instead they prefer to rely on standards, experience and communications with other programmers [4].

Research into reading comprehension in other domains has found ink annotations are beneficial for improving understanding and retention of what has been read [5–9]. Annotating with a pen is a subconscious task: the reader annotates without interrupting the reading process [10]. Freehand ink annotation is more effective than the types of text annotation available in word processors [10, 11].

Code today is written and read directly on computers with programmers very rarely printing out code. As ink annotations provide benefits in other domains we speculate

J. Abascal et al. (Eds.): INTERACT 2015, Part II, LNCS 9297, pp. 177–194, 2015.
DOI: 10.1007/978-3-319-22668-2_15

that programmers may also get similar benefits. But there are no previous studies on how programmers annotate code. We hypothesize that when programmers read code they would add ink annotations if it were possible.

In this study we investigated how experienced programmers annotate when reading for comprehension. We analyze the nature and the purpose of their annotations. From this we compare their annotations to standard programming environment support and suggest ways in which ink annotations in Integrated Development Environments (IDEs) may be beneficial.

2 Background

2.1 Annotations in Reading

Annotations allow the reader to form a conversation with the text as they read. This changes the reader from being a passive receiver to being actively engaged in the text. Being actively engaged in the text helps people to understand and remember information [10]. Previous studies have found that annotation is intrinsically linked with reading [12, 13].

Annotations provide value over notes on separate paper by being close to the associated context. Having the context is especially useful when the reader is working through problems as it allows the reader to visualize the annotations and context simultaneously [14]. There is also evidence that the physical location of the annotations assists with finding them again [15]. People know roughly where they added something and will quickly flick to that location. In addition hand-written annotations stand out from the underlying context which also makes it easier to find them again [15].

Another strategy where annotations assist is with learning information by summarizing it [7]. The reader summarizes key points as they occur. Then during re-reading these summaries reduce the amount of re-reading needed. Annotations provide value over separate notes because they are situated in context. If the reader needs more information they can directly refer to the original text [16].

Different styles of reading have different characteristics. Some styles are characterized by a requirement of non-linear reading [17]. Non-linear reading is more demanding cognitively than linear reading. Readers need to hold what they are reading in memory at the same time as finding the next relevant section. In this context annotations reduce the mental workload [17].

2.2 Nature of Programs

Program code is very different from other forms of text. Program code is non-linear: programmers do not read sequentially from start to finish but trace paths through the code instead [18]. Code is highly structured: it follows rules that allow computers to execute it, and conventions that help programmers read it. Large programs are often split across multiple files. Tracing the flow of execution can involve swapping between multiple files and/or multiple locations within each file.

These differences make reading code very different from other forms of reading. Tools can compensate for these differences by assisting programmers with comprehending code. Examples of support provided by tools include alternate visualizations (e.g. graphs and diagrams), debugging support and syntax highlighting [19, 20]. There are also tools that allow people to draw diagrams and annotate their code [21, 22].

Navigating through code is an important task for programmers. If they cannot find a section then they may not understand the surrounding code. Navigation tools are common in IDEs. Common tools include search, project explorers and quick links. These tools can help programmers quickly find their way around code.

There are very few tools that allow freeform digital ink annotations and none of these are commercially available [18, 21, 23, 24]. This limitation means there have been no studies on how programmers might annotate code if there were no imposed constraints. There is some anecdotal evidence that annotations may help with navigation and comprehension [23]. There is also anecdotal evidence that the number of annotations decreases as the size of the program increases [25].

3 Methodology

For this study we observed experienced programmers reading a program they had not seen before. We investigated the following research questions:

- How do experienced programmers annotate code?
- Why do they annotate code?

3.1 Participants

There were thirteen participants in this study. All were programmers with at least five years programming experience and they all had prior experience with C#. They all currently worked in a commercial environment or had previously worked in one. All but one participant was male.

Experienced programmers were chosen for this study to reduce any confounding effects due to prior experience. It is assumed they know how to read code already.

3.2 Task

The participants were all asked to read a short block of program code. Prior to the reading task they were told that they would have to explain how the code works to a less experienced programmer. The rationale given for the task was that a new graduate would be taking over responsibility for the code and needed assistance with understanding the code. They were not told to annotate the code. If asked they were told they could do anything with the paper; including drawing or writing on it.

The code consisted of six files from a larger system written entirely in C#. The code in this study was responsible for initializing communications between modules in the

system. All relevant code was provided. The code was printed for the participants to read. The printed code used 14 pages of paper.

Paper was chosen as a medium to reduce the effects any particular tool might have. There are some tools that allow freeform annotations on screen but these are currently not robust and programmers are not used to using them [18, 21, 23, 24]. In contrast reading on paper is a common task for most people.

Each participant was given a copy of the code, some extra blank pieces of paper, a highlighter and four different colored pens (red, blue, black and green).

The participant was seated at one side of a table. At the other side of the table a video camera was setup to record what the participant was doing. The camera was focused on the participant's hands and the pieces of paper. The investigator sat behind the camera and ensured it was correctly focused and adjusted as needed during the task. The investigator also took notes during the task.

3.3 Procedure

Each participant performed the task in a controlled environment. Prior to the reading task each participant was welcomed to the lab, had the process explained to them (including gaining consent) and filled in a short questionnaire on their programming and reading background.

Each participant was given 45 min to read the program code. They could finish earlier if desired. While reading each participant was video recorded and observed. We answered questions about the process (e.g. can I write on this paper?) but not about the code (e.g. what does this class do?) At the end of the reading time their paper was collected and the purpose of the study was explained.

After the reading task there was a short interview about their annotations. This involved looking at the annotated paper and asking the participant to explain why they had made each annotation. This included the importance of the annotation and how it fitted into their approach to understanding the code.

3.4 Analysis

After each participant finished the data was coded. We went through each page and counted the number of annotations and recorded the following attributes about each:

- Location
- Type of annotation
- Reason for adding annotation.

Location was a choice from within code or left | right | top | bottom of code. This was based on where the majority of the annotation was.

The type of annotation was based initially on the annotations described by Marshall [26]. When a new annotation type was found the list was expanded to include it. After coding the list was reviewed and similar types of annotations were combined.

During the interview the investigator asked the participant the general purpose of each annotation. This was then summed up in a few words (e.g. "question", "highlight

for later", "possible bug", etc.) The reasons where then consolidated into a list of basic reasons based on the participant's intention when the annotation was added.

These attributes were then summarized and collated with the answers from the questionnaires.

4 Results

The majority (9 out of 13) of the participants currently used C#. All of these participants reported using it daily. The other participants all had some experience with C# and were confident that they could read a short program in it.

Most participants reported reading code on a daily basis. Reasons for reading included debugging, learning, reviewing other people's code and general development (e.g. extending an existing codebase). Most participants read their own code on a daily basis. The two participants who did not read their own code on a daily basis did read other people's code at least weekly. While most participants reported reading code on screen daily only one participant reported regularly printing off code for reading. All other participants said they rarely printed code to read on paper.

During the study 12 of the participants added annotations or notes. Only one participant (P05) did not add any annotations or notes. In the post-observation interview we asked about this. He reported he never makes any annotations and very rarely takes notes when using pen and paper. His reason for this was it was the way he was trained. He was excluded from the remaining analysis as an outlier.

4.1 How Programmers Annotate

In total the twelve participants added 267 annotations (see Table 1). Ten participants added annotations on the code; the other two participants only wrote on separate pieces of paper. The average number of annotations added by each participant was 33. The number of files annotated ranged from one to all six. On average each participant added eight annotations per file.

The following types of annotations were identified:

- Underline: a line drawn underneath text
- Scratch-out: a line drawn through text
- Highlight: a line drawn through text with the highlighter
- Enclosure: a circled block of code
- Margin bar: a vertical line, typically drawn in the margins
- Brace: a } like annotation spanning multiple lines of code
- Connector: a joining line between two elements (e.g. a block of code or another annotation) without an arrow
- Arrow: as above but with an arrow head
- Text: one or more characters (e.g. letters, numbers, punctuation marks)
- Drawing: a diagram or picture
- Dot: a small mark or dash.

Table 1. General Annotation Details

Participant	Number of annotations added	Number of files annotated	Average annotations per file
P01	44	6	7
P02	36	5	7
P03	21	2	11
P04	76	6	13
P06	31	4	8
P07	12	4	3
P08	0	–	n/a
P09	29	2	15
P10	0	–	n/a
P11	5	2	3
P12	9	1	9
P13	4	2	2

These annotations were classified into five categories:

- Single line: these span a single line and are associated with code or other text. Underlines, highlights and scratch-out all fall into this category.
- Multiple line: these span multiple lines of code but do not contain explicit meaning beyond their location. Enclosures, margin bars and braces fall into this category.
- Connector: these join two or more items together. One common use for a connection annotation is to associate a text annotation with a segment of code. Arrows and connectors fall into this category.
- Complex: these have explicit meaning associated with them, although maybe only to the original annotator. Text and drawings are both in this category.
- Attentional: these are indirect signs of the participant's attention. They are temporary and not intended for future re-use. Dots fall into this category.

In addition some annotations consist of multiple annotation types: these are compound annotations. The participant added these together at the same time and considered them as a single annotation. A common example is a connector and text together.

The annotations added by each participant are shown in Fig. 1. While there are no common patterns among the participants in this study there are two general trends. Four of the participants (P01, P03, P04 and P07) preferred mainly single-line annotations. Four of the remaining participants (P02, P09, P11 and P12) preferred mainly complex annotations. Of these annotations they are predominately text notes.

The majority of the annotations were added within the code (65 %). The next most common location is to the right of the code (24 %) and then to the left or top of the code (6 % and 5 %). Only one participant added an annotation below the code – this involved copying some lines of code from the next page. Figure 2 shows where each participant added the annotations.

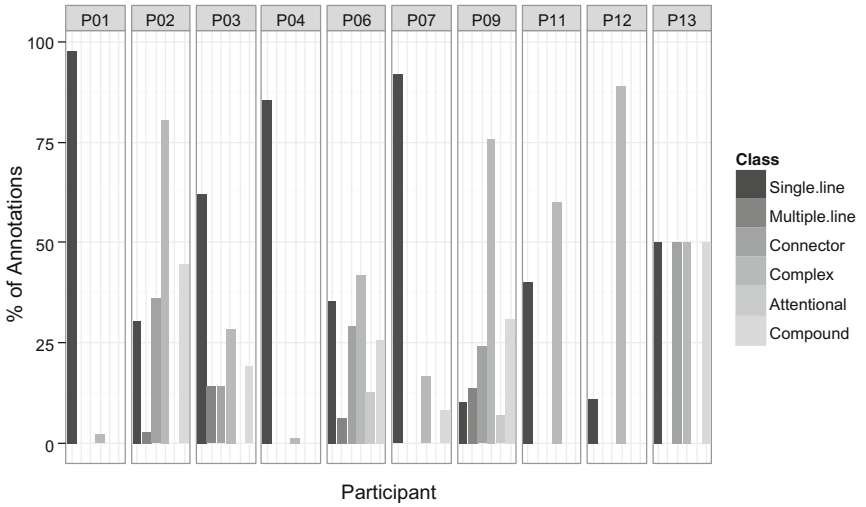

Fig. 1. Percentages of annotation types added on the source by classification per participant. Percentages are percentage of total annotations added by the participant. Compound annotations are a separate classification – therefore the totals may add to more than 100 %.

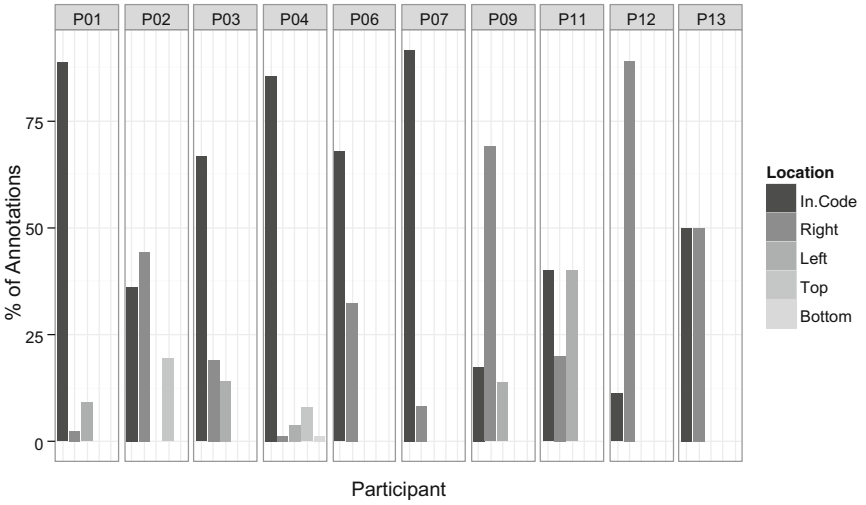

Fig. 2. Percentage of annotations added to each location per participant. Percentages are percentage of total annotations added by the participant.

There is a relationship between the classification of annotation and its location. Single line and attentional annotations mainly occurred within the code. Multiple line and connector annotations mainly occurred to the left or the right of the code. Complex annotations occurred at any location.

4.2 Why Programmers Annotate

Initial data entry identified 39 different reasons for adding the annotations. During the analysis we identified that many reasons were very similar in nature. Therefore similar reasons were grouped together resulting in eleven basic reasons which fall into four categories (see Table 2).

We grouped these basic reasons into four categories based on the main intent. The first category of reasons, navigation, is for helping the participant find their way through the code. These typically speed up the process of re-finding information. The second category, working information, is where the participant is recording things they have found about the code. This is information they think will be useful later on. The third reason, information for sharing, is similar but intended for someone else. The final reason, other, is for the more uncommon annotation reasons.

Navigation Annotations. The most common reason for adding an annotation is to emphasize something for future use. This marks an item that the participant is thinking of returning to later – even if they never do. Common examples of this are: underlining a method call or name, circling a section of interesting code and highlighting where variables are defined. These annotations were generally single-line annotations.

There were two participants who extensively annotated for this reason (P01 and P04). They both said they were trying to build a model of the code. P01 said he was trying to replicate the solution explorer in Visual Studio (the solution explorer lists all the classes and methods in a set of code files). He went quickly through the code initially to highlight most of the method names. Then he went through the code a second time using the highlighted method names to trace what was happening. In contrast P04 used the highlighted method names as a form of backtracking. When he started in a method he would highlight the method name. Later he could easily return to this method by scanning through the paper for the highlight.

Other participants who emphasized for future reference were more selective. They only emphasized the elements of interest. There were two main motives: emphasizing hard-to-find elements and commonly referenced elements. Both motives were to help the participant re-find them later.

Another reason is to add a reference. These annotations link two or more sections of code together. The participants added these when they thought they would need to move quickly between two segments of the code. One example of this reason is matching letters to the side of the code. The participant was able to quickly scan through the code and find the matching reference as needed. These annotations were mainly text like a symbol or the name of the method. During the interview participants said the references were not always used but when used were valuable especially for hard-to-find segments of code.

By emphasizing code structure the participant was trying to make it easier to move between different files. In all cases the participants added these at the top of the first page in the file. These annotations were either text or a highlight. The highlight or text was always positioned so the participant could see it quickly.

Emphasizing a significant feature had two categories: important code and hard-to-find code. These differ from other navigation reasons in one significant aspect: the

Table 2. Consolidated list of reasons why annotations were added.

Reason	Description	Number	Percentage[a]
Navigation			
Emphasize for future	The annotation marks a feature of the code to be reviewed later	117	44 %
Add reference	A reference to a different section in the code	21	8 %
Emphasize code structure	Highlight a structural element in the code	17	6 %
Emphasize significant feature	A section of the code that needs further investigation	9	3 %
Working Information			
Record working notes	Inline notes on what is happening in the program	36	13 %
Record question	A query about something in the code	36	13 %
Correct previous annotation	A correction or update to a previous annotation	6	2 %
Information Sharing			
Record a needed change	An area of the code that needs to be changed	11	4 %
Emphasize example	An example of another section of code is called	2	1 %
Other			
Unknown	The participant was unsure as to why they added the annotation	6	2 %
Attentional	An incidental mark as a result of the user's attention	6	2 %
Total Number of Annotations:		**267**	

[a]Due to rounding these percentages do not add to 100 %.

participant expects they will implicitly remember the rough location of the annotation. The annotation speeds up finding the location again. One participant mentioned this is like adding a post-it note to the page.

Working Information Annotations. We identified three reasons for annotations that were for recording information. These reasons were: working notes; questions and corrections. All three reasons were an attempt to reduce the cognitive workload while reading.

Working notes are a description about what is happening in the code. The participant would add a working note when they figured out something and wanted to remember it later. The notes are a way of offloading this information onto paper to reduce their mental workload. These are typically text comments although they might only make sense to the writer. Occasionally a participant would return to their working notes and make a correction if they realized a comment was incorrect.

A question annotation is where the participant wants to find out some more information. Questions are different from working notes because the participant is seeking an answer. In contrast working notes are what the participant already understands. Unlike working notes question annotations have a wide variety of forms. They could be as simple as an underline or asterisk next to some code right up or a complete written question. During the interview we asked for more details about these question annotations including what the underlying question was and whether it was answered later.

We found three types of question: implementation, functional requirements and language features. A question on the implementation is why the code was implemented a certain way. For example, one participant was interested in why there were two methods with the same name but different parameters. A question on the functional requirements was when the participant saw something that was not clear if it matched the described functionality of the system. For example, one participant questioned whether a method had the correct logic. Finally a question on a language feature was when the participant saw something they did not understand about the language or base libraries. For example, one participant queried about how a lambda was coded.

An implementation question was typically answered by the participant in their reading when they understood the code better. But only one participant updated these annotations with the answer. Most of the participants said they just remembered the information in their head. The annotation reminded them of the question and answer. Questions about the functional requirements were sometimes answered when the participant read other parts of the code. None of the participant updated these annotations although some added a reference to the question. One participant described these as questions to take to the business owner about whether the code is correct. The final type were questions that would be answered by researching the feature: looking online, asking someone or looking up some form of reference materials. None of the participants updated these annotations during the session.

A correction annotation is where the participant has returned to a previous annotation and updated it. Corrections always involved scratching out part or all of an annotation. In two instances the participant also added additional alternate text.

Information Sharing Annotations. Participants also added annotations to record something that needed changing. Examples of these annotations included hard-coded values, unclear variable or method names and bad coding practices. Most were enclosures; there was only one text annotation in this category. The participants who added these annotations stated they only had to look at the code to remember why they added the annotation. This implies that these annotations are a form of offloading from memory onto the paper.

The least common reason was emphasizing an example. Both example annotations were where the participant thought the code was a very good example for a junior programmer to see.

Other Reasons. Attentional annotations are incidental marks of the participant's current focus. During the task the participants often pointed to the code with the pen currently in hand. Occasionally the participant would make contact with the page. During the interview the participants who made these annotations mentioned they were accidental and they did not care about these annotations.

Finally, we added a reason of unknown. These are when the participant did not remember why the annotation was added. However this was uncommon: they only failed to remember the reason six times out of 267 annotations.

4.3 Separate Notes on Paper

In addition to making annotations on the code nine of the participants made notes using separate pieces of paper. Three of these participants used two pieces of paper while the rest used a single piece. Only one of the participants added notes to both sides of the piece of paper. This resulted in a total of thirteen pages of notes. Of these thirteen pages four were text only. The remaining pages contained diagrams and text (see Fig. 3).

The most common usage for notes is class diagrams and flow diagrams. In class diagrams the participant is drawing a graphical representation of the classes. There were six class diagrams drawn. A flow diagram shows a flow of either information or control through the code. There were five flow diagrams drawn.

The class diagrams typically included the interfaces defined and relationships to other classes. These were mainly drawn at an overview level. Only one of the diagrams contained property and method members. In addition these members were a subset of the full members for the class. One of the diagrams was text only; the rest contained text and graphics elements. These ranged from simple arrows connecting classes and/or interfaces to boxes similar to UML format. Two of the diagrams contained textual notes in addition to the diagram (see Fig. 3). These were notes explaining the purpose of some of the classes.

The flow diagrams were more varied. Three diagrams worked at a high level (i.e. server and client) and showed some of the flows between these components. One of these had the relevant classes written for each component, another had the methods and the third was a conceptual diagram. The other flow diagrams were all based at the class level. They had some of the classes in the code and the messages that were passed between them. Two flow diagrams had textual notes. One set of notes listed the data structure being passed around; the other listed some background information to the data flows.

Two pages contained notes of findings (see Fig. 3). These findings were information the participant thought relevant to share with the junior programmer. Both of these pages were text only. They were both grouped lists. The first line in each group was the heading and the remaining lines were indented.

We matched these pages with two of the basic reasons: working information and information for sharing. During the interviews the participants stated the information was either for themselves or for sharing with the junior programmer. Three of the participants started writing the notes with the intention of sharing. The other six participants intended the notes to be only for themselves; these notes were working information. Three of these participants stated that they would share their notes with someone else. However they stated they would not directly share the working notes as they did not consider the information complete. They expected to either sit down with the person reading it or to revise it into a form that could be shared.

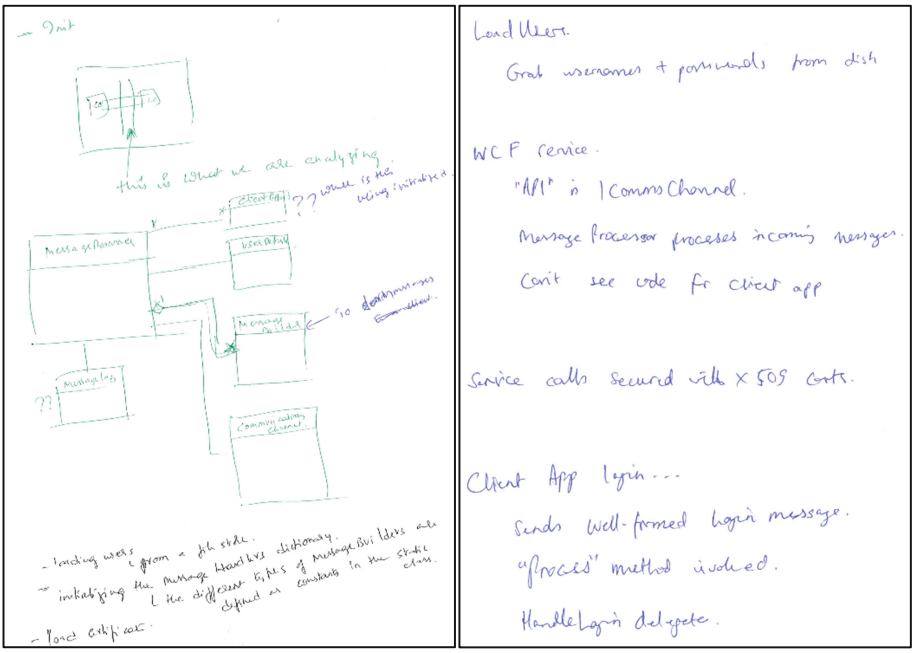

Fig. 3. Examples of note pages: left page (from P12) shows a class diagram with added notes; right page (from P09) shows textual notes for sharing with another programmer.

The time when the participants wrote the notes also varied widely. Two of the participants started writing notes very soon in the reading process. These participants did not make any annotations on the code; instead they used the separate pages for writing information. Five of the participants started writing notes later in the process. These participants used a combination of annotations on the code and separate note pages. The remaining two participants annotated the code as they read. Near the end of their reading they then went through and wrote notes on the separate paper.

The reason for the notes was linked to when the participant started writing them. The two participants who wrote the notes at the end of the session wrote the notes specifically for the junior programmer. In contrast the two who started writing at the start of the session wrote the notes for their own understanding. The remaining participants were split between writing for the other person and for themselves. These were also the participants who changed the intention of the notes.

4.4 Other Observations

In addition to the observations about the annotations there were some other interesting patterns. These are not specific to annotating code but reading code on paper in general.

One of the first things eleven of the participants in the study did was split the code into the separate files (most files contained a single class, one contained two). They

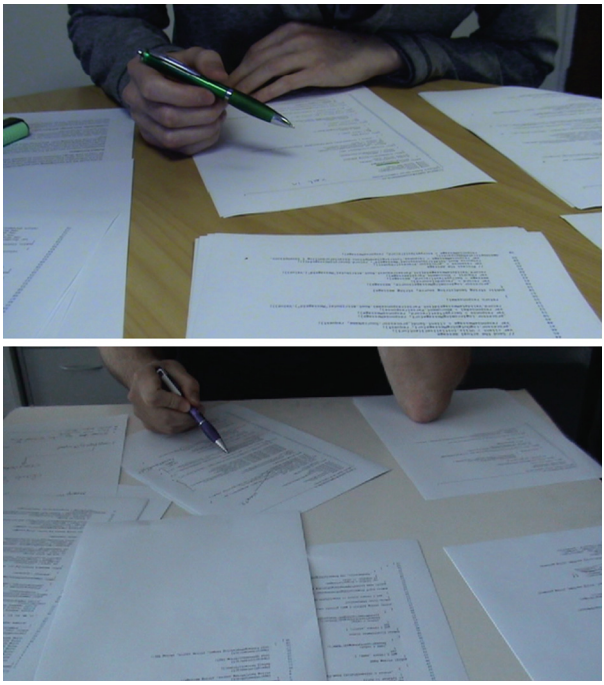

Fig. 4. Examples of paper spread around the desk.

were originally given a single stack of paper containing all the code: they took this stack and leafed through to find how many different files there were. Then they separated the files into separate stacks of paper.

All the participants spread these stacks of paper around the desk (see Fig. 4). They reported a variety of reasons why they did this: to keep the files separate, to gain an overview of the code, to know where the various files are (for moving between them). Some of the participants combined this separating of pages with annotations. For example, P02 wrote the class names on the top of the first page in each class so he could quickly find them again.

Seven of the participants combined the separation of pages with spatial positioning. Once they had separated the piles they tended to keep them in the same positions. They would pick up a pile to read or search through it and then return it to the same position. For four of these participants this was a conscious behavior: they had deliberately made the decision to do this when separating the piles. One participant (P11) had even arranged the location of the piles based on their sizes. The underlying rationale was the bigger piles were more likely to be used. So he placed them in locations where he could quickly reach them. The other three participants used spatial positioning without having made a conscious decision on it. One participant, P05, stated that he did not originally intend to keep them in the same locations but midway through the read task found it was easier to find things if he returned them to the same location.

5 Discussion

In this study all but one participant added annotations to the code or wrote them on a separate piece of paper. This was expected as the environment was impoverished compared to what is available in most IDEs. There are a number of different reasons given for adding annotations.

The frequency of some annotation types are similar to what has been reported in other studies (e.g. [27]). Underlines and highlights had similar frequencies to what has been reported before. This may be because they are so simple and easy to add.

One annotation type that is more frequent in this study is compound annotations. The majority of the compound annotations consist of a connector plus another type of annotation. We posit this is because annotations need to be associated with specific lines of code. If the participant did not need a precise anchor they would not use a connector but would be simply a sidebar as is seen in prose annotation.

The majority of annotations were added to help the participant navigate the code. This is similar to what other research has shown for when programmers read code [1, 2, 4]. Our participants used a range of strategies to mark specific sections of code (such as highlights and margin bars) and also cross-reference marks to indicate connections between code on different pages. For example most participants highlighted some method names. This selective highlighting, while similar to IDE color syntax highlighting, is more specific in that the participants highlighted only some method names.

Another common form of navigation annotations was for backtracking. Again this is functionality that IDEs already provide. But IDE navigation support is generic where annotations are specific. The readers annotate to focus on what they are interested in and to reduce the workload by place marking. This type of annotation is seen in other studies [15, 17] but the importance of navigation in code comprehension makes it more critical for code understanding. Digital ink annotation in IDEs could be a valuable aid to support code navigation.

Computers can, and do, generate a lot of navigation information but they do not know what the reader is interested in. Therefore the reader may take false routes to find what they need, sometimes even backtracking to revisit previously scanned code [4]. While annotations cannot help find unvisited code they can help reduce the clutter of backwards navigation. A combination of computer generated navigation links and user ink annotation may be optimal.

The other main reason was for recording information. Offloading information to paper reduces the amount of cognitive workload the reader needs to perform [15]. P08 mentioned that one of his challenges was trying to remember everything. This reduction in cognitive load and abstraction that ink annotation affords is consistent with other studies [10, 11] suggesting that ink annotation inside code editors could offer similar benefits.

Annotations on the code and on the separate pieces of paper both help to reduce the cognitive load. However there appear to be different benefits between the two. Annotations on the code were shorter and more cryptic than on the separate notes. In contrast the diagrams on the separate paper would be more difficult to add to the code because of space constraints and also they tend to be an overview rather than location

specific. However the separate notes were artifacts that the participants mentioned they would want to keep. These results corroborate the ideas of Lichtschlag, Spychalski and Bochers [22] who suggested that hand-drawn sketches assist programmers in orientating in a codebase.

The spreading out of the paper on the table is an interesting observation that could benefit from further research, in particular is the importance of spatial positioning. There are tools that allow graphical representations for code (e.g. [19, 20]). These tools should maintain the spatial positioning of the elements to help programmers remember where things are. Large screen or multiple screen systems may also be beneficial by providing a larger space for programmers to work. The programmer could move the code files into positions that they find useful. The current file would be moved into a prominent location for work. Then when finished it would be returned to its previous location. If the programmer needs it later they can easily find where it is based on its location. We are not aware of any research into this type of programmer support.

This study is part of a larger project looking at utilizing freeform digital ink annotations in an IDE. In code editing tools the main way to record information is by adding comments in the code. Previous studies comparing textual annotations vs. freeform annotations found people prefer freeform annotations [10, 11]. O'Hara and Sellen suggested the most likely reason for this is freeform annotations do not interrupt what the reader is thinking about [10]. Therefore freeform annotations may be useful for storing information against the code without reducing the reader's capacity for comprehending what they are reading. Second, annotations are not limited to text. Several of the participants drew diagrams of how they understood things. Studies have found that diagrammatic annotations provide value by keeping the diagram close to its context [14]. Finally, annotations stand out from the text. Freeform annotations look very different from the underlying document. Many of the participants in the study were able to quickly find things by just scanning through the code and looking for the annotations. This, when combined with spatial memory, allowed them to easily find something they had previously written.

6 Limitations

First, participants were given a specific task to do (read the program code for understanding). A different task may have resulted in different annotation patterns (e.g. marking an assessment, adding unit tests). Second, most of the participants used IDEs that provide a variety of tools to assist with understanding. Some of these tools might be replicable on paper (e.g. an index of all the method locations). If these tools were available then the participants may have used different types of annotations. However we contend that the selective nature of annotations may, in some circumstances, be better than generic functionality. Third, the programming language may have an effect on the types of annotation. Some languages are more difficult to read: a programmer reading program code in these languages may add different types of annotations. Finally, this study only had 13 participants. Including more participants might make some of the patterns observed more obvious. However this study does provide some insight on how annotations could be useful in an IDE.

7 Conclusions and Future Work

This study investigated how and why experienced programmers annotated code on paper. Previous work has shown it is possible to combine freeform ink annotations within a code editor [21, 24] but there was no evidence of if, how and why programmers would annotate code with digital ink.

The results of this study indicate that somewhere to record information would assist programmers comprehend what they read. The ability to annotate on code with freeform ink may be useful if the functionality were available. Separate blank pages may also be beneficial; especially if there is some way to link the notes to the code (see [22]).

This study shows that programmers use annotations to assist with navigation, record information as working notes and to remember information for sharing. While some of this functionality is currently available in code editing tools, user specified particular navigation paths, which were frequently used in our study, are not supported in IDEs. Furthermore, freeform ink annotations provide an alternate avenue that reduces the cognitive work needed when reading code. This is because ink annotations are quicker and easier to add compared to text-based annotations, allow greater expressiveness and stand out from the code because they are visually distinct from the text.

It would be interesting to compare how participants used an IDE for the same task. Comparing the two toolsets would allow a comparison of what functionality is currently available and what could be provided in future.

Future lines of investigation include:

- How would programmers annotate code if an IDE supported freeform annotation?
- How can freeform annotations assist navigation?
- How could spatial layout of files assist comprehension?

Acknowledgment. We would like to thank all the participants in our study for time they gave us.

References

1. Sillito, J., De Voider, K., Fisher, B., Murphy, G.: Managing software change tasks: an exploratory study. In: International Symposium on Empirical Software Engineering 2005, pp. 23–32 (2005)
2. Singer, J., Lethbridge, T., Vinson, N., Anquetil, N.: An examination of software engineering work practices. In: CASCON First Decade High Impact Papers, pp. 174–188. IBM Corporation, Toronto (2010)
3. Sellen, A.J., Harper, R.H.R.: The Myth of the Paperless Office. MIT Press, Cambridge (2002)
4. Maalej, W., Tiarks, R., Roehm, T., Koschke, R.: On the comprehension of program comprehension. ACM Trans. Softw. Eng. Methodol. **23**, 1–37 (2014)

5. Fowler, R.L., Barker, A.S.: Effectiveness of highlighting for retention of text material. J. Appl. Psychol. **59**, 358 (1974)
6. Hynd, C.R., Simpson, M.L., Chase, N.D.: Studying narrative text: The effects of annotating vs. journal writing on test performance. Reading Res. Instruction **29**, 44–54 (1989)
7. Simpson, M.L., Nist, S.L.: Textbook annotation: an effective and efficient study strategy for college students. J. Reading **34**, 122–129 (1990)
8. Wolfe, J.L., Neuwirth, C.M.: From the margins to the center: the future of annotation. J. Bus. Technical Commun. **15**, 333–371 (2001)
9. Ball, E., Franks, H., Jenkins, J., McGrath, M., Leigh, J.: Annotation is a valuable tool to enhance learning and assessment in student essays. Nurse Educ. Today **29**, 284–291 (2009)
10. O'Hara, K., Sellen, A.: A comparison of reading paper and on-line documents. In: Proceedings of CHI 1997, pp. 335–342. ACM, New York (1997)
11. Morris, M.R., Brush, A.B., Meyers, B.R.: Reading revisited: evaluating the usability of digital display surfaces for active reading tasks. In: Proceedings of TABLETOP 2007, pp. 79–86. IEEE (2007)
12. Adler, A., Gujar, A., Harrison, B.L., O'Hara, K., Sellen, A.: A diary study of work-related reading: design implications for digital reading devices. In: Proceedings of CHI 1998, pp. 241–248. ACM, New York (1998)
13. Hong, M., Piper, A.M., Weibel, N., Olberding, S., Hollan, J.: Microanalysis of active reading behavior to inform design of interactive desktop workspaces. In: TABLETOP 2012, pp. 215–224. ACM, New York (2012)
14. Jackel, B.: Item differential in computer based and paper based versions of a high stakes tertiary entrance test: diagrams and the problem of annotation. In: Dwyer, T., Purchase, H., Delaney, A. (eds.) Diagrams 2014. LNCS, vol. 8578, pp. 71–77. Springer, Heidelberg (2014)
15. Johnson, M., Nadas, R.: Marginalised behaviour: digital annotations, spatial encoding and the implications for reading comprehension. Learn. Media Technol. **34**, 323–336 (2009)
16. Glover, I., Xu, Z., Hardaker, G.: Online annotation – Research and practices. Comput. Educ. **49**, 1308–1320 (2007)
17. Crisp, V., Johnson, M.: The use of annotations in examination marking: opening a window into markers' minds. Br. Educ. Res. J. **33**, 943–961 (2007)
18. Priest, R., Plimmer, B.: RCA: experiences with an IDE annotation tool. In: Proceedings of CHINZ 2006, pp. 53–60. ACM, New York (2006)
19. Bragdon, A., Reiss, S.P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola Jr., J.J.: Code bubbles: rethinking the user interface paradigm of integrated development environments. In: Proceedings of ACM/IEEE International Conference on Software Engineering, pp. 455–464. ACM, New York (2010)
20. DeLine, R., Bragdon, A., Rowan, K., Jacobsen, J., Reiss, S.P.: Debugger canvas: industrial experience with the code bubbles paradigm. In: Proceedings of the 2012 International Conference on Software Engineering, pp. 1064–1073. IEEE Press (2012)
21. Sutherland, C.J., Plimmer, B.: VsInk: integrating digital ink with program code in visual studio. In: Proceedings of AUIC 2013, pp. 13–22. Australian Computer Society, Incorporation (2013)
22. Lichtschlag, L., Spychalski, L., Bochers, J.: CodeGraffiti: using hand-drawn sketches connected to code bases in navigation tasks. In: Proceedings of VL/HCC 2014, pp. 65–68. IEEE (2014)
23. Chen, X., Plimmer, B.: CodeAnnotator: digital ink annotation within Eclipse. In: Proceedings of AusCHI 2007, pp. 211–214. ACM, New York (2007)

24. Lichtschlag, L., Borchers, J.: CodeGraffiti: communication by sketching for pair programmers. In: Adjunct Proceedings of ACM Symposium on User Interface Software and Technology, pp. 439–440. ACM, New Ork (2010)
25. Plimmer, B.: A comparative evaluation of annotation software for grading programming assignments. In: Proceedings of AUIC 2010, pp. 14–22. Australian Computer Society, Incorporation (2010)
26. Marshall, C.C.: Annotation: from paper books to the digital library. In: Proceedings of ACM International Conference on Digital Libraries, pp. 131–140. ACM, New York (1997)
27. Marshall, C.C., Brush, A.J.B.: Exploring the relationship between personal and public annotations. In: Proceedings of Digital Libraries, 2004, pp. 349–357. ACM Press, New York (2004)