# Obstacles to Comprehension in Usage Based Reading

David J. A. Cooper, Brian R. von Konsky, Michael C. Robey and David A. McMeekin
Curtin University of Technology
Department of Computing
GPO Box U1987
Perth, Western Australia 6845
david.cooper@postgrad.curtin.edu.au

## Abstract

*Usage Based Reading (UBR) is a recent approach to object oriented software inspections. Like other Scenario Based Reading (SBR) techniques it proposes a prescriptive reading procedure. However, the impact of such procedures upon comprehension is not well known, and consideration has not been given to established software cognition theories. This paper describes a study examining software comprehension in UBR inspections. Participants traced the events of a UML sequence diagram through Java source code while thinking aloud. An electronic interface collected real-time data, allowing the identification of "points of interest", which were categorised according to issues affecting participants' performance. Together with indicators of participants' cognitive processes, this suggests that adherence to UBR scenarios is non-trivial. While UBR can detect more critical defects, we argue that a re-think of its prescriptive nature, including the use of cognition support, is required before it can become a practical reading technique.*

## 1 Introduction

Usage Based Reading (UBR) is a relatively new technique to support the inspection of object oriented systems. It promises to allow the detection of more critical defects than traditional Checklist Based Reading (CBR) by using use case scenarios to guide the inspection effort. While some recent studies have found UBR to be more efficient and effective than CBR in specific circumstances [16, 15, 20], other studies have not demonstrated such advantages [5, 13].

Owing to its infancy, UBR best practice remains largely a matter of speculation. As with Scenario Based Reading (SBR) [2] more generally, it relies on inspectors following a rigidly defined procedure. Such a requirement seems at odds with theories of software cognition, which suggest that comprehension strategies are employed on an opportunistic basis [12, 11, 18]. Moreover, although UBR requires inspectors to match inspection artefacts against use case scenarios, little is known of how this matching is or should be done. Theoretical concerns regarding this have been expressed, and solutions including tool support and the insertion of visual cues into artefacts have been proposed [10, 19]. The effects of unassisted UBR on comprehension, however, have not been explored.

In this paper we describe a study in which the ability of inspectors to follow a UBR scenario is examined in isolation. Given one use case scenario and a system of 193 executable lines of code, participants were asked to identify those lines, and the order thereof, that would be executed in the scenario. Participants were also asked to think aloud during this exercise. Six issues affecting participants' ability to locate the correct lines were identified from data collected. Data generated from the think aloud process indicate digressions from the scenario and large variations in participants' cognitive processes. Such phenomena would limit the suitability of a rigid, prescriptive approach to reading software arefacts.

## 2 Background

UBR is one of a family of reading techniques known as Scenario-Based Reading (SBR), in which inspectors are asked to traverse the documents in a rigidly defined manner [2]. In the case of UBR, the artefacts under inspection are verified against each step of each use case scenario in turn. Usage Based Reading is the name given to the technique by Thelin *et al.* [17]. The use-case reading approach investigated by Dunsmore *et al.* [5], although not originally described as UBR, is founded on the same premise and for each use case entails essentially the same activity.

Although in this paper we describe both approaches as variants of UBR, differences do exist in their respective

**Table 1. Previous studies comparing UBR and CBR.**

| Study | Participants | Use case prioritisation | Explicit state verification | Sequence diagrams | Source code | Reported efficacy |
|---|---|---|---|---|---|---|
| Dunsmore *et al.*[5] (2003) | 69 (3rd year undergraduate students) | none | yes | yes | yes | lower than CBR |
| Thelin *et al.*[16] (2003) | 23 (master's students) | prior | no | no | no | higher than CBR for critical faults |
| Thelin *et al.*[15] (2004) | 62 (master's students) | prior | no | no | no | higher than CBR |
| Winkler *et al.*[20] (2004) | 131 (undergraduate and graduate students) | prior/during inspection (different groups) | no | no | no | higher than CBR |
| McMeekin [13] (2005) | 54 (36 undergraduate students and 18 industry professionals) | none/prior (different groups) | yes | yes | yes | no significant difference |

descriptions and in the studies evaluating them. Use case prioritisation, whereby use cases are ranked beforehand in order of importance to the user and then inspected in that order, was found by Thelin and colleagues to increase the ability of UBR to detect critical defects. This was not investigated by Dunsmore and colleagues. Furthermore, the study by Dunsmore *et al.* was conducted using source code and sequence diagrams, whereas that of Thelin *et al.* concerned textual and diagrammatic design and requirements documents, and represented use case scenarios in task notation rather than UML.

Table 1 lists prior studies where UBR variants have been compared to CBR, the *status quo* reading technique, in which inspectors find defects by consulting a generic list of potential issues. Some comparisons of UBR and CBR have found the former to be more efficient and/or effective in detecting critical defects (and in some cases all types of defects). Overall, however, the evidence is inconclusive.

Possible explanations for the differences in the results of these various studies include the artefacts involved, instructions given to participants, participants' level of experience, whether use cases were prioritised, or other factors. The effects of a number of these factors upon UBR remain largely unknown. In particular, no qualitative empirical research has been done to uncover the issues an inspector might encounter, consciously or otherwise, in the midst of inspecting artefacts for an individual use case scenario.

The existence of comprehension issues in UBR can be inferred from proposed models of software cognition. Although several cognition models exist [14], many of them propose that two or three distinct comprehension strategies are used opportunistically. For instance, a combination of top-down and bottom-up strategies is proposed by Mills [12, p. 148], Letovsky [11] and von Mayrhauser and Vans [18]. A selection mechanism must exist to choose between these strategies, the operation of which depends on the level of familiarity with the system, the quality of the available documentation, and the person involved.

A rigidly specified procedure for traversing and reading a set of artefacts, such as exists in UBR and more generally in SBR, conflicts with notions of opportunistic comprehension strategies, and thus could effectively override the selection mechanism. Although outcome of the selection mechanism is ultimately unpredictable, we presume experience is an important factor. To override that mechanism would therefore be to impinge upon the ability of experienced inspectors to use their experience. The studies in Table 1 reporting higher UBR efficacy compared to CBR could have had different outcomes if participants in those studies had been industry professionals.

Moreover, comprehension strategies themselves incorporate elements of iteration and unpredictability that are not generally accounted for in SBR. This could undermine the ability of inspectors to follow a use case scenario in the first place.

Cognitive aspects of inspections in general have been explored, using a technique called *protocol analysis* [6] whereby participants in an experiment are asked to think aloud. Each participant's verbalisations are recorded, and form a *verbal protocol*. This is coded according to a predefined coding scheme in order to extract objective, qualitative data on the participant's cognitive processes. For

instance, with data collected using the technique Letovsky proposed that software comprehension proceeds via *inquiry episodes*, which are individual instances of the application of existing knowledge and experience to comprehending a software system [11]. Also using protocol analysis, Hungerford *et al.* [9] found that inspectors who switch their attention rapidly between different artefacts tend to be more effective at detecting defects [9]. This lends support to the use of UBR, which requires that inspectors frequently switch their attention back and forth between a use case scenario and the artefacts under inspection. Kim *et al.* [10] found that such context switching can be done more effectively if visual cues are used to assist inspectors in finding related information spanning multiple artefacts. Visual cues can thus be a form of *cognition support* for inspections.

To provide additional cognition support to UBR, Walkinshaw *et al.* [19] propose an automated method to map behavioural specifications to source code using dependence graphs [8], which have previously been applied to the creation of tools to support CBR [1, 3]. Their technique finds a path of execution through source code based on several *landmark methods* derived from the specification. Other method calls and the execution of conditional constructs are inferred from this information, by taking *slices* of a dependence graph. To evaluate this approach, they compared their tool's functionality to an existing set of cognition support criteria [14].

The actual effects of UBR on comprehension are nonetheless not well understood, and this is the principal contribution of this paper.

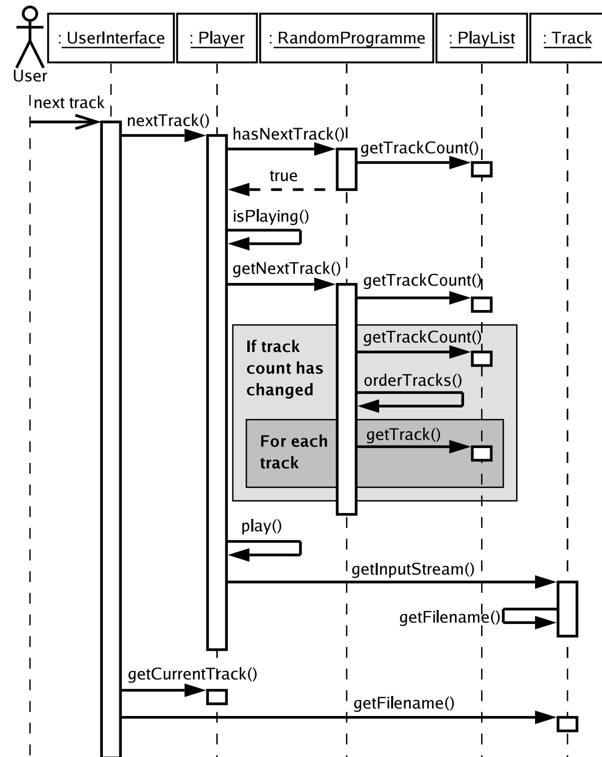


**Figure 1. The UML sequence diagram shown to participants, representing the scenario in which the *next track* option is chosen while tracks are being played in a random order. The shading emphasises nesting levels.**

## 3 Methodology

Our study was conducted to qualitatively examine the problem of tracing the events of a use case scenario through source code. Our approach employed two data collection methods: an online interface through which participants completed an inspection-related task, and thinking aloud. These methods produced distinct but related sets of results.

Data collected through the online interface facilitated analysis of the actual steps participants took towards completing the task, as discussed in Section 3.4. Our use of protocol analysis described in Section 3.5 relied on this data, but also provided insight into how comprehension of the system proceeded.

### 3.1 Participants

Ten participants were involved: five industry professionals, three university-employed graduates and two undergraduate students. Industry experience ranged from zero to five years of professional practice. No incentives were offered to participants other than experience gained. The study was approved by the Curtin University Human Research Ethics Committee.

### 3.2 Materials

The system under inspection was a simple Java-based audio player controlled from a command-line interface. One use case scenario was presented in the form of a UML sequence diagram, shown in Figure 1.

Excluding comments, blank lines and brace only lines, the source code consisted of 330 lines of code in seven files. However, to avoid confusion regarding what constituted a line of code, only executable statements and field initialisers were numbered. In this scheme, there were 193 numbered lines of code.[1]

Participants had not previously seen any of the artefacts presented.

[1]The source code and other artefacts presented to participants are available at http://se-research.cs.curtin.edu.au/postgrad/cooperdj/aswec3/.

**Table 2. An excerpt of the input updates recorded for participant 8. Each row represents one update. Times of the updates are relative to the start of the exercise. For brevity, unchanging parts have been replaced with "[cut]".**

| time | input update |
|---|---|
| 18:40 | `UserInterface.java 5-19`<br>`UserInterface.java 18-19`<br>`UserInterface.java 22,27,31-32`<br>`Player.java 27`<br>`Programme.java 3`<br>`Playlist.java 15`<br>`Player.java 28,35-38,28-29,30?,31-32,` |
| 20:59 | `[cut]`<br>`Player.java 28,35-38,28-29,30?,31-32`<br>`Programme.java` |
| 21:08 | `[cut]`<br>`Player.java 28,35-38,28-29,30?,31-32`<br>`Programme.java 5,6` |
| 25:20 | `[cut]`<br>`Player.java 28,35-38,28-29,30?,31-32`<br>`RandomProgramme.java 5,6` |
| 25:25 | `[cut]`<br>`Player.java 28,35-38,28-29,30?,31-32`<br>`RandomProgramme.java` |
| 25:35 | `[cut]`<br>`Player.java 28,35-38,28-29,30?,31-32`<br>`RandomProgramme.java 8` |

### 3.3 Procedure

Each participant completed an inspection-related activity individually in the presence of a researcher. Participants were asked to list, in order of execution, each line of code that is or might be executed in the scenario. Such an instruction imposes an rigid ordering on the exercise, because one line's participation in the scenario cannot be determined until all previous lines in the scenario have been found.

The task was undertaken through an online interface. Participants added entries, each consisting of a class name and line number or range, to an input box for lines of code they determined should be executed. Question marks and asterisks were added where execution of a line was conditional or repeated (within the scenario) though this information was not eventually used. The times at which the input box was modified were recorded automatically in a database. The *decision time* for each update — the time elapsed since the previous update — was also recorded. No time limits were imposed on participants.

Table 2 shows an example of the data collected in this way (the importance of which is discussed further in Sections 4). Participants were asked to enter lines in a given format, though the format of the actual input data varied. A parser was developed to convert each input update into a list of $< class, line\ no. >$ pairs, accounting for the format variations, correcting class name misspellings through a manually constructed look-up table and ignoring partially complete entries.

During the exercise, participants were asked to think aloud and were prompted to "keep talking" if silent for more than 30 seconds. They were each given two short training tasks to familiarise them with this process.

Audio recordings were made independently of the online interface. These were synchronised with the data captured through the online interface by comparing mouse clicks in the former to mouse events recorded in the latter. This synchronisation allowed the protocol analysis process to use contextual information from the input data.

### 3.4 Input Data Analysis

Having translated participants' input updates into a common format, the data was then analysed in a partially automated process. This is briefly illustrated in Figure 2, in which an example set of input updates are given in the leftmost table. These are a simplification of the format in Table 2, though decision times are explicitly shown.

Using an edit distance algorithm the updates were broken down into individual line additions and deletions. In Figure 2 "line 3" was added in the first update, "line 5" in the second, "line 4" and "line 3" in the third and "line 10" in the fourth, in which "line 4" was also deleted.

These additions and deletions were reconstructed into a single list of all lines a participant indicated, where multiple updates pertaining to the same line of the solution (e.g. where a line was listed and later de-listed) were aggregated into a single list entry. The reconstructed list preserved the times at which lines were indicated and their intended order of execution, including where the same line was indicated at multiple points in the solution. The bottom, centre table in Figure 2 illustrates this reconstruction. The lines connecting it to the leftmost table show the input updates from which each table row was derived. The decision times shown are the sum of those of the associated updates.

Two investigators independently created lists of the lines executed, and a model solution was constructed after agreement was reached. In Figure 2 an example model solution is shown instead in the top, centre table.

Each participant's reconstructed list was matched against this model solution. Errors were recorded where mismatches occurred. Each matching line, and the first of each continuous sequence of mismatching lines, was assigned a *solution index*. The solution indices map each participant's input to the model solution, and thus allow different participant solutions to be compared and aggregated. The right-

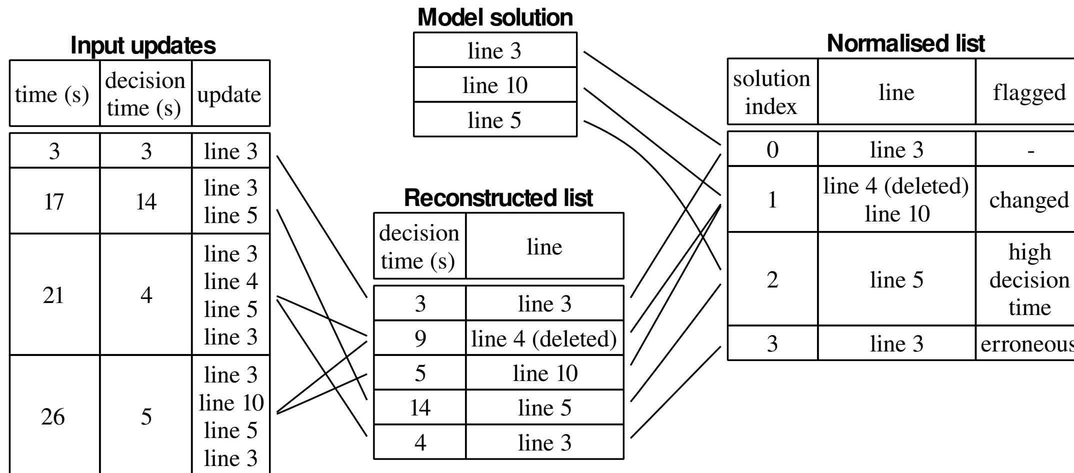


**Figure 2. An outline of the process used to transform a participant's input into a comparable, normalised form, using an example model solution and input.**

most table in Figure 2 represents the result of matching the example reconstructed list against the example model solution. Lines connecting it to those two tables indicate where comparisons occurred.

*Points of interest* were designated where lines, having been assigned solution indices, had one or more of the following characteristics:

- The line deviated from the model solution. Several small deviations were discounted as trivial, where it was plausible that a participant did nonetheless understand the path of execution. These included some instances of line duplication (as shown at the top of the first row in Table 2), two adjacent lines listed in reverse order, or a missing method call before the correctly-listed contents of that method.

- The line was associated with a high decision time, relative to other times for the same participant. High decision times were considered to be those more than 0.5 standard deviations above the mean decision time for a given participant.

- The line had been deleted (and possibly later re-listed).

### 3.5 Protocol Analysis

Participants' verbal protocols were used to give two more broad indicators regarding the extent of the correspondence between their cognitive processes and the activity's instructions. The *cognitive overlap* indicator gauges the variability of the cognitive process between participants, by determining the overlap between participants' verbalised thoughts at any given solution index. The *digression* indicator gauges the extent to which participants were able to concentrate on each step of the use case scenario in turn, by determining the relevance of their thoughts to the line additions or deletions at each input update. The role of these two indicators in providing guidance on reading technique design is discussed in Section 5.

As in any objective analysis of verbal data, the protocols were coded. The relative simplicity of the coding scheme allowed this to be done directly from the audio recordings, without the need for transcription or segmentation. However, an excerpt of one transcript is presented in Figure 3 for illustration purposes.

The period(s) of time spent by each participant at each solution index, as captured by the online interface, were used to determine where to listen in the audio recordings. Each reference a participant made verbally to a method, method call, constructor, field or variable in the source code, or a message, object or either of the two shaded regions in the sequence diagram was recorded. References to standard Java or JDK constructs were not coded.

Examples of references include:

- *Player.nextTrack()*, which indicates the *nextTrack()* method in the *Player* class,

- *Player.nextTrack():programme.hasNextTrack()*, which indicates a call site of the method *hasNextTrack()* accessed through the *programme* variable within the *nextTrack()* method, or

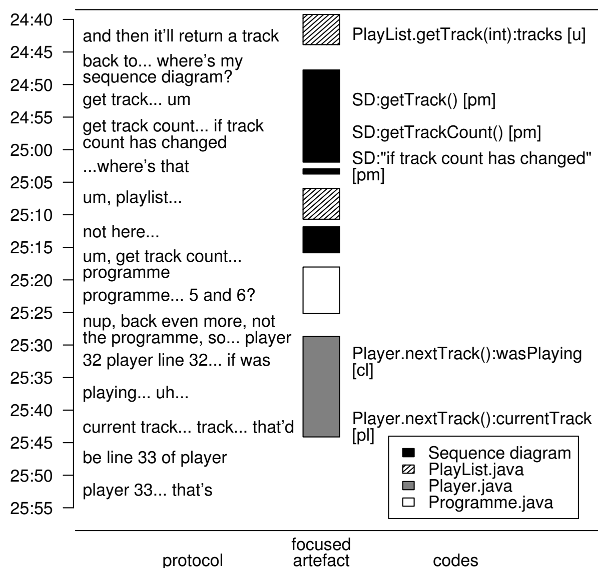


**Figure 3. An excerpt of a participant's verbal protocol at one solution index, the artefacts focused on at the time, and the codes assigned.**

- *SD:nextTrack()*, which indicates the *nextTrack()* message as passed in the sequence diagram.

An example of the coding process is given in Figure 3. Ultimately, a set of references was produced at each solution index for each participant.

Participants often omitted one or more of the first components in each reference, and contextual information had to be relied upon to resolve any ambiguity. Such contextual information included the artefact currently or previously focused on and the method currently being inspected. The former was recorded by the online interface, and the latter inferred from the current or previous solution index.

Although no duplicate references existed within each set, it was possible for some references to be counted for more than one solution index in cases where a single input update added or deleted more than one line. This also occurred where a reference was made at the same time as an input update, and so was counted for both that update and the next one. This duplication is not problematic, however, because no cumulative analysis was done on the references.

A second coding scheme was then superimposed on the first to give an indication of the relevance of each reference to each solution index. A reference to a field, variable, method or message appearing at the current line was labelled *cl*. Failing this, if it appeared on the previous or next line of either the model solution or the source code, it would be labelled *pl*. The label *cm* was otherwise given for references to or within the same method, or *pm* for references to or within the previous or next method in the model solution, on a line-by-line basis. Similarly, *cc* was given for references to or within the current class, or *pc* for those to or within the previous or next class in the model solution. If no other categories were applicable, a reference was considered unrelated and labelled *u*.

Each reference received one of the above *proximity codes* depending on its proximity to the current solution index line of code. These codes were assigned by comparing each reference to the current solution index in the model solution and selecting the first applicable code in order.

**Table 3. Overall characteristics of each participant's solution.**

| participant | total time | mean decision time | model solution coverage | line mis-matches |
|---|---|---|---|---|
| 1 | 54 min | 35 sec | 42% | 74% |
| 2 | 35 min | 28 sec | 85% | 37% |
| 3 | 55 min | 45 sec | 33% | 83% |
| 4 | 35 min | 44 sec | 63% | 25% |
| 5 | 32 min | 43 sec | 62% | 20% |
| 6 | 43 min | 32 sec | 73% | 27% |
| 7 | 42 min | 38 sec | 67% | 55% |
| 8 | 46 min | 37 sec | 85% | 54% |
| 9 | 39 min | 36 sec | 63% | 46% |
| 10 | 28 min | 37 sec | 40% | 64% |
| mean | 41 min | 39 sec | 61% | 48% |

# 4 Results

## 4.1 Input Data

An overview of the characteristics of each participant's solution is given in Table 3. Considerable variation exists in both model solution coverage (33–85%), and the proportion of line mismatches (20–83%). Model solution coverage indicates the proportion of the lines of the model solution that also appear in each participant's solution. Line mismatches indicate the proportion of the lines in each participant's solution absent from the model solution.

On average, 41 minutes were spent by each participant on the task. This is high, given that a real inspection is recommended to last for no more than two hours [7] and would involve several use case scenarios. However, the requirement for participants to explicitly note each line of code would have added to the time spent.

**Table 4. Overall scale and effect of the comprehension issues identified.**

| issue | points of interest | affected participants (/10) | mean decision time |
|---|---|---|---|
| scenario start misidentification | 13 | 10 | 204 sec |
| accidental omission | 24 | 9 | 33 sec |
| polymorphism | 8 | 7 | 120 sec |
| method misidentification | 8 | 6 | 37 sec |
| context switching | 23 | 10 | 98 sec |
| condition evaluation | 28 | 9 | 76 sec |

By comparing points of interest derived from each participant's solution, we identified six common issues. (In some cases, as a result of an earlier omission, participants did not have a chance to confront parts of the source code in which certain issues commonly arose.) These issues are summarised in Table 4.

- *Scenario start misidentification.* No participant's solution began at exactly the point defined by the model solution. Four listed lines occurring prior to this point, and six omitted at least some subsequent to it. Anecdotal evidence from one participant suggests that there may have been confusion over the term "command-line based" as given in the instructions. The term referred to the fact that the system instituted its own command-line, not that it operated necessarily from an existing command-line. A consequence of the latter interpretation may have been that the initial construction sequence of the system was assumed to be part of the scenario, as indicated by five participants, whereas in fact the sequence diagram made no mention of it.

- *Accidental omission.* Method omissions, where participants listed method calls but not the corresponding method contents, were a common occurrence, accounting for 14 points of interest. In another instance in the system, the Java *Arrays.sort()* method is called with a customised *Comparator* object, a result of which is that in the scenario the object's *compare()* method is called from a standard Java class. No participants identified this method. Three more points of interest were attributable to the omission of the last one or two statements in a method.

- *Polymorphism.* At the method call *programme.getNextTrack()*, seven participants initially indicated that the *getNextTrack()* method of the *Programme* class was executed. The sequence diagram, however, specifies that the call is actually made to an instance of the subclass *RandomProgramme*. Five participants eventually realised this (perhaps because many of the messages on the sequence diagram were passed only from the subclass's method).

  The challenge posed to participants by polymorphism is evident in the raw input data in Table 2. In the third input update (at 21:08), participant 8 has incorrectly listed lines 5 and 6 of *Programme.java*, which represent the *getNextTrack()* method. This begins to change in the fourth update (at 25:20), more than 4 minutes later. After a few more seconds (at 25:35), line 8 of *RandomProgramme.java* is correctly listed. Although the lines ultimately identified by the participant were correct, the high decision time involved and the fact that lines were deleted identified this instance as a point of interest.

- *Method misidentification.* In two cases of method overloading, six points of interest were the result of participants indicating the wrong method. One more point of interest arose from a participant listing the *getNextTrack()* method in place of *hasNextTrack()*, and another where one accessor method was listed in place of another that happened (as an implementation detail) to return the same value.

- *Context switching.* High decision times were recorded in 19 instances where method calls took place, and in 4 cases after methods returned. We attribute this to the time and effort associated with locating a line not adjacent to the previous line, often in another class. 7 of the 19 high decision times associated with method calls were for the *first* method call.

- *Condition evaluation.* Seven points of interest resulted from participants incorrectly evaluating *if* statement conditions (though four were likely the result of a previous error of the same kind). These were either false negatives or false positives. A further ten high decision times were recorded for the first line inside an *if* statement. Similar evaluation is required to determine whether an exception is thrown at a given point in the scenario. None were included in the model solution, but 11 points of interest resulted from participants listing them. These all reflect the difficulty of manually evaluating dynamic conditions in a static context.

  Determining whether a boolean condition is satisfied or if an exception occurs, in the context of a specific

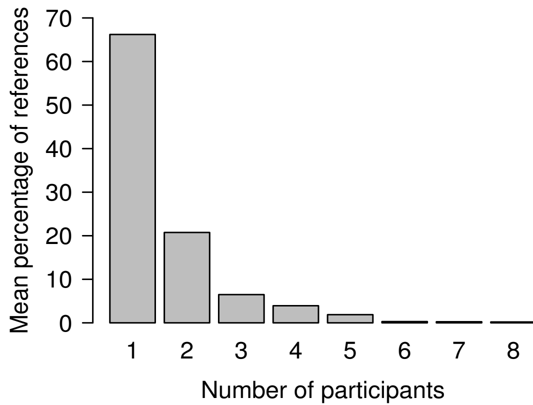


**Figure 4. Overlap in participants' references, averaged across solution indices. Two thirds of references were made by only one participant.**

use case scenario, requires the consideration of various constraints imposed by the scenario and the source code. For example, if a message represented on the sequence diagram corresponds to a method call inside an *if* statement, the *if* condition must be true in that scenario. Likewise, if throwing an exception would prevent one or more messages from being passed, it cannot occur within the scenario.

In addition to the above issues, a small number of other points of interest remain unclassified. These include four instances where both branches of an *if* statement were listed, two where no logic appeared to connect one listed line to the previous line, and one where a high decision time was recorded for no apparent reason.

### 4.2 Verbal Data

The two types of coded data described in Section 3.5 were first analysed at the solution index level.

For each reference made at each solution index, the number of participants to whom the reference was attributed was counted. Averaged over all solution indices, this comprises the first indicator mentioned in Section 3.5. The results are presented in Figure 4. The first bar indicates the mean percentage of references attributed to only one participant (any participant), the second the number attributed to two different participants, and so on. Thus, on average two thirds of references were made by only one participant.

The proximity codes, assigned to each reference as an indication of relevance, were further categorised according to the issues identified in Section 4.1. For each issue, the number of references that fell into each proximity category were averaged over all participants for all relevant solution indices. This data is presented in Figure 5. References made in the absence of identifiable issues were generally very close to the current line. Unrelated references were proportionally more numerous in cases involving scenario start misidentification, accidental omissions, context switching and condition evaluation. Moreover, the number of references overall was higher where issues were identified.

## 5 Discussion

The model solution coverage for participants in our study, as given in Table 3, points to an important problem. Coverage of inspection artefacts — the proportion of an artefact actually subject to human inspection — in UBR is known to be less than 100%, since the available set of use case scenarios is itself unlikely to correspond exactly to all the source code. Dunsmore *et al.* [5] cited anecdotal concerns from participants in their study to this effect. However, much of the source code that *is* modelled in a use case scenario is also liable to be effectively left out of a UBR inspection. Participants in our study only identified between 33% and 85% of the relevant code.

The effects of these omissions may be mitigated by involving multiple inspectors, as originally envisaged by Fagan [7]. Although doing so remains a good idea, this is not a perfect solution because the issues encountered by one inspector could apply to others as well. Many of the issues discovered were not manifested randomly, but occurred at the specific points in the use case scenario. For instance, seven out of ten participants in our study encountered difficulty with the same instance of polymorphism. Hence, in UBR inspections polymorphism may well go unnoticed or misunderstood by several different inspectors, and thus large amounts of relevant code could escape inspection altogether. It is desirable, therefore, to provide cognition support for inspectors targeting such problematic constructs.

A second, related problem is evident in the proportion of line mismatches given in Table 3. Inspectors need not be completely accurate in their analysis of the path of execution. For instance, small deviations to examine the effects of exception handling may help rather than hinder inspection performance. Large deviations, however, will defeat the purpose of UBR. In our study, on average almost half the lines inspected by participants were deviations from the scenario. The coded verbal data presented in Figure 5 does indicate that small digressions from the current point in the scenario occurred even when no identifiable issues arose. When issues did arise participants digressed further from the scenario, despite the fact that many errors were themselves in close proximity to the model solution.

Here too, Dunsmore *et al.* expressed their concern that "more participants using this technique deviated from the

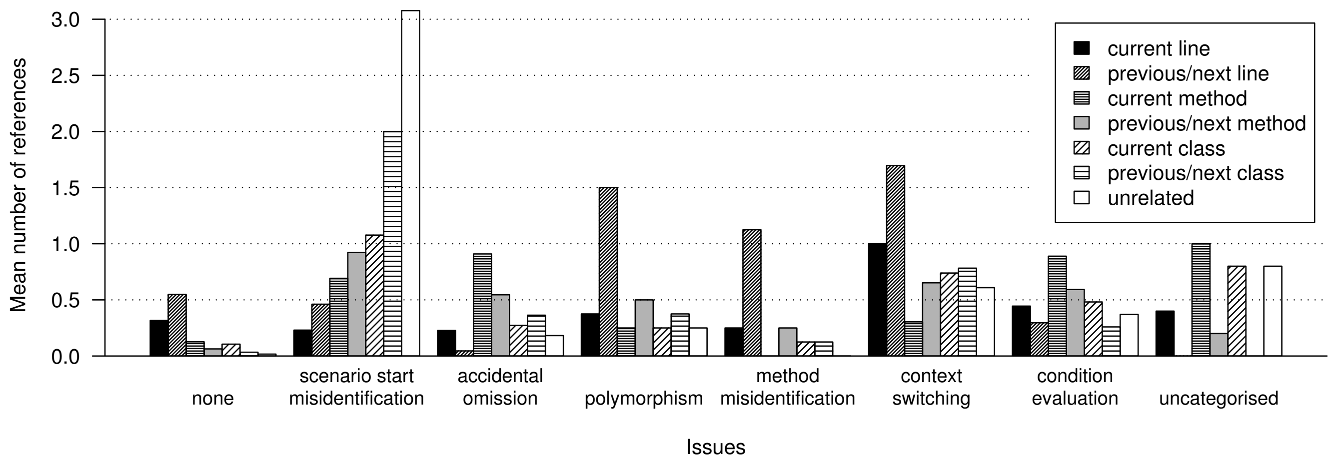


**Figure 5. The mean number of references in each proximity category for each issue.**

recommended application". In this case, the risk is that too much time will be spent inspecting non-critical aspects of the software, to the exclusion of its most important functionality. In addition, although switching between artefacts contributes to understanding a software system [10, 9], more time spent switching contexts (or resolving other comprehension issues) represents more time likely not spent inspecting anything at all.

Such problems may not be limited to UBR. SBR more generally employs a rigid inspection procedure. In light of the variability in participants' verbal references presented in Figure 4, it is questionable whether following a rigid procedure for traversing the inspection artefacts leads to a similarly predictable understanding of the system. We argue that the need to comprehend the system properly overrides the narrow scope of the procedure, detracting from its effectiveness as a means to focus inspectors' attention on specific aspects of the system.

UBR does have at least one important advantage over CBR — an ability to detect more critical defects [16, 15, 20]. However, it is arguably the perspective of the system that UBR gives inspectors, not the procedure as a whole, that can be credited with this accomplishment. As suggested by Denger *et al.* [4], other reading techniques including CBR may be re-engineered to provide the same perspective as variants of SBR without the encumbrance of the scenario itself. A usage-based CBR technique could thus be designed, in which checklist questions direct inspectors to review parts of the inspection material associated with each use case scenario in turn.

The provision of cognition support could also help to recreate UBR's use case perspective. For example, visual cues such as investigated by Kim *et al.* [10] could be used in place of explicit instructions to follow the scenario. Such cues could be inserted into artefacts (temporarily, prior to inspection) to highlight the beginning of a scenario, method call sites, called methods themselves, method scopes and where conditional blocks of code (*if*, *switch* and *catch* blocks) are or might be executed. The last may otherwise be inferred anyway where method calls take place. If UBR inspections are to be carried out using a software tool directly, hyperlinks connecting each call site to the set of methods potentially called in the scenario (in polymorphic cases there will be more than one) could be used to further reduce context switching time. These would address the issues described in the previous section.

Deriving the visual cues themselves would require an approach such as that detailed by Walkinshaw *et al.* [19]. Their dependence graph technique could largely automate the identification of source code corresponding to a use case scenario. With such support, inspectors would be able to focus their attention to a greater extent on finding defects rather than determining (and often failing to determine) the path of execution. Moreover, inspectors would be free to make digressions as they see fit, while being consciously aware that they are digressions rather than direct examinations of the relevant use case scenario. In combination with the use of visual cues, therefore, such tool support would provide a powerful means to overcome many of the comprehension issues identified in this study. By bridging the gap between source code and use case scenarios, it could also help to ensure that scenarios (in task notation or UML) are kept up to date when the system is modified, and hence could make UBR applicable in situations where it would not otherwise have been.

## 6 Conclusion

The advent of object orientation and the resulting dispersion of related information among multiple artefacts sug-

gests that new reading techniques are required for inspections. UBR promises not only to encourage focus switching between artefacts to increase inspector awareness of their interrelationships, but to give priority to the detection of critical defects. These arguments in favour of UBR remain valid, and the evidence indicates that the technique is effective at least in certain circumstances.

Our data suggests, however, that challenges face unassisted UBR and SBR. These arise from difficulties inherent in understanding a set of artefacts while executing a rigid procedure for traversing them. Theories of software cognition imply that more flexibility is needed, and this is supported by the verbal data presented in this paper. While UBR's perspective is valuable, other mechanisms for providing it should be sought out to facilitate the adoption of the technique as a practical alternative to CBR.

Although several studies comparing UBR to CBR have been done, more research is needed to determine how efficacy varies with different kinds of artefacts (e.g. source code vs. design diagrams), different variations on the instructions and different levels of prior experience with UBR. In particular, further investigations should be conducted to quantify the effects of visual cues and dependence graph-based tool support on the technique. Results from such future studies could help establish a theoretical basis upon which UBR can be refined and adapted for use in real-world software projects.

## Acknowledgements

Thanks are due to Steven Webb for assisting in the construction of the model solution used in our analysis, to the ten participants for donating their time, and to the anonymous reviewers for their comments.

## References

[1] P. Anderson, T. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Transactions on Software Engineering*, 29(8):721–733, Aug 2003.

[2] V. R. Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3–12, 1997.

[3] D. J. A. Cooper, M. W. Chan, M. Harding, G. Mehra, P. Woodward, B. R. von Konsky, and M. Robey. Using dependence graphs to assist manual and automated object oriented software inspections. In *Proc. of the 2006 Australian Software Engineering Conference (ASWEC'06)*, pages 262–269, Apr 2006.

[4] C. Denger, M. Ciolkowski, and F. Lanubile. Investigating the active guidance factor in reading techniques for defect detection. In *Proc. of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*, pages 219–228, Aug 2004.

[5] A. Dunsmore, M. Roper, and M. Wood. The development and evaluation of three diverse techniques for object-orientated code inspection. *IEEE Transactions on Software Engineering*, 29(8):677–686, Sep 2003.

[6] K. A. Ericsson and H. A. Simon. *Protocol Analysis: Verbal Reports as Data*. MIT Press, revised edition, 1993.

[7] M. E. Fagan. Design and code inspections to reduce errors in program development. *IEEE Systems Journal*, 15(1):182–211, 1976.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 3(9):319–349, 1987.

[9] B. C. Hungerford, A. R. Hevner, and R. W. Collins. Reviewing software diagrams: A cognitive study. *IEEE Transactions on Software Engineering*, 30(2):82–96, Feb 2004.

[10] J. Kim, J. Hahn, and H. Hahn. How do we understand a system with (so) many diagrams? Cognitive integration processes in diagrammatic reasoning. *Information Systems Research*, 11(3):284–303, Sep 2000.

[11] S. Letovsky. *Empirical Studies of Programmers*, chapter 5, pages 58–79. Ablex Publishing Corporation, 1986.

[12] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice*. The Systems Programming Series. Addison-Wesley Publishing Company, 1979.

[13] D. A. McMeekin. A comparison of code inspection techniques. Honours thesis, Curtin University of Technology, Perth, Australia, Nov 2005.

[14] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[15] T. Thelin, C. Andersson, P. Runeson, and N. Dzamashvili-Fogelström. A replicated experiment of usage-based and checklist-based reading. In *Proc. of the 10th International Symposium on Software Metrics (METRICS'04)*, pages 246–256, Sep 2004.

[16] T. Thelin, P. Runeson, and C. Wohlin. An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering*, 29(8):687–704, Sep 2003.

[17] T. Thelin, P. Runeson, C. Wohlin, T. Olsson, and C. Andersson. How much information is needed for usage-based reading? - a series of experiments. In *Proc. of the 2002 International Symposium on Empirical Software Engineering (ISESE'02)*, pages 127–138, 2002.

[18] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, Aug 1995.

[19] N. Walkinshaw, M. Roper, and M. Wood. Understanding object-oriented source code from the behavioural perspective. In *Proc. of the 13th International Workshop on Program Comprehension (IWPC'05)*, pages 215–224, May 2005.

[20] D. Winkler, M. Halling, and S. Biffl. Investigating the effect of expert ranking of use cases for design inspection. In *Proc. of the 30th EUROMICRO Conference (EUROMICRO'04)*, pages 362–371, Sep 2004.