An Inter-Rater Reliability Analysis of Good's Program Summary Analysis Scheme

Pauli Byckling, Marja Kuittinen, Seppo Nevalainen and Jorma Sajaniemi
Department of Computer Science
University of Joensuu, Finland

Keywords: POP-I.A. learning to program, POP-II.B. program comprehension, POP-V.B. protocol analysis

Abstract

In computer science education and research into the psychology of programming, program summary analysis has been used to characterize mental models of novice and expert programmers and to measure learning outcome of programs and programming concepts. This paper reports an investigation where three raters used Good's program summary analysis scheme consisting of two independent classifications of program summary segments: information types and object description categories. The problems in using the scheme as well as differences between the raters were recorded and analyzed. The findings indicate that by improving the scheme and its documentation, most of the observed inter-rater differences can be avoided. The only open problem concerns making the distinction between descriptions of data and activities in cases where the specific words that are used, or the abstractness of expression may affect raters' interpretation of the information type.

Introduction

In computer science education (CSE) and research into the psychology of programming (PoP) one central question is the problem of measuring the quality of learning and comprehension of programming concepts. There is no universally agreed-upon measure of comprehension (Dillon and Gabbard, 1998) but the following methods have been applied: written or verbal tests on recall, recognition (e.g., multiple-choice questions), and relationships between concepts; problem solving exercises (i.e., programming tasks); essays evaluated by their correctness or by qualitatively analyzing students' mental models; and interviews. In the context of programming, essays may also be program summaries, i.e., free-form explanations of programs.

Program summary analysis has been used in CSE and PoP to characterize mental models of novice (Corritore and Wiedenbeck, 1991) and expert (Pennington, 1987) programmers attaining high levels of comprehension, to characterize mental models of students capable of reusing program code (Hoadley et al., 1996), to describe how mental models depend on underlying programming paradigm (Good, 1999), and to evaluate learning outcome in novice programmers (Kuittinen and Sajaniemi, 2003). These studies suggest that program summary analysis can be utilized in revealing students' mental models and that the contents of the mental models can be used to characterize the quality of comprehension.

The basic idea of program summary analysis is to ask subjects to provide a free-form explanation, or summary, of a program just studied. By omitting detailed instructions about the form of the summary, subjects' own preferences guide the selection of information in the summary and a wide variation in the responses is usually achieved. The program summary methodology avoids the problems of false positive results often associated with binary choice questions, and the difficulties in designing sensitive and reliable multiple choice questions (Good and Brna, 2003). In program summary analysis, the interest is not in the correctness of the summary; the abstraction level and the types of information are more important characterizations of the mental model than a detailed memorization of the program code.

In order to analyze program summaries, some analysis scheme must be used. The earlier studies used Pennington's scheme (Pennington, 1987) based on information types and levels of detail. Later, Good

(1999) devised another scheme based on information types, but more finely-grained and fully specified than in Pennington's scheme, and object descriptions, which is a restricted version of Pennington's level of detail. Hoadley et al. (1996) used a simple classification based on abstraction level similar to Good's object descriptions but consisting of fewer categories. von Mayrhauser and Lang (1999) and O'Brien et al. (2001) have developed schemes for coding program comprehension protocols but these schemes cannot be used to analyze program summaries.

Good's scheme has been used in some studies (Good, 1999; Good and Brna, 2003; Sajaniemi and Kuittinen, in press) but its replicability has not been analyzed (Good and Brna, 2003), e.g., there is no knowledge of its inter-rater reliability nor of reasons for possible differences among raters. This paper reports an investigation where three raters used Good's scheme to analyze real program summary data. The problems in using the scheme as well as differences between the raters were recorded and analyzed.

The rest of the paper is structured as follows. The following section gives an introduction to Good's scheme. The next two sections describe the investigation and discuss its results. Finally, the last section contains the conclusions.

Good's Scheme

Good's program summary analysis scheme has been described in two documents. Good's PhD thesis (Good, 1999) gives detailed instructions including Coding Manuals and coding examples whereas an article by Good and Brna (2003) describes various categories of the scheme with few examples. We will now give a short summary of Good's scheme; see the above references for more exact definitions.

Good's program summary analysis scheme consists of two independent classifications of program summary segments. The first is based on *information types* (IT), i.e., what kind of information about the program a statement reveals. The other classification is based on *object description categories* (ODC) that look at the way individual objects are described in summaries. The interesting items are different in these two classifications and consequently program summaries are segmented differently for these two purposes.

For the purposes of IT coding, program summaries are segmented to short passages consisting of a subject and a predicate (either of which may be implied). The classification comprises eleven categories:

- **Function (FUN):** The overall aim of the program, described succinctly.
- **Actions (ACT):** Events occurring in the program described at a lower level than *Function*.
- **Operations (OPE):** Small-scale events which occur in the program, such as tests, assignments etc.
- **State-high (SHI):** Describes the current state of a program when a condition has been met (and upon which an action is dependent).
- **State-low (SLO):** A lower-level version of *State-high*. *State-high* describes an event at a more abstract level than *State-low* which usually describes the direct result of a test on a single data object.
- **Data (DAT):** Inputs and outputs to programs, data flow through programs, and descriptions of objects and data states.
- **Control (CON):** Information having to do with control structures and with sequencing.
- **Elaborate (ELA):** Further information about a process, event or data object which has already been described. This also includes examples.
- **Meta (MET):** Statements about the participant's own reasoning process.
- **Unclear (UNC):** Statements which cannot be coded because their meaning is ambiguous or uninterpretable.
- **Incomplete (INC):** Statements which cannot be coded because they are incomplete.

IT categories are related in terms of level of granularity. *Function* describes the highest level of abstraction, i.e., the purpose of the program which can be described with no reference to how it is achieved; *Actions* and *State-high* describe in an abstract manner the way that the program works; *Operations* and *State-low* correspond to single lines of code. Figure 1 gives an example of IT coding.

This program checks a basketball players height from the list given.	Actions
If the height of the player is over 180	State-high
Then he is selected for the team.	Function
Once there are five players	State-high
the program is terminated.	Control

Figure 1: An example of IT coding (Good, 1999).

The program wants **all marks over 65** listed	Domain
and **all marks over 66** will pass	Domain
the exam.	Domain
The output will state	Program
the mark	Domain
and whether **the person** has passed.	Domain

*Figure 2: An example of ODC coding (Good, 1999). The coded object descriptions are marked in **boldface**.*

For the purposes of the ODC coding, the interesting objects must first be selected and only then the program summaries can be segmented. ODC coding is applied to data objects only, and program summaries are segmented so that there is exactly one data object per segment. The classification comprises seven categories:

- **Program only (PON):** References to items which can occur only in the program domain.
- **Program (PRO):** References to objects, which could be described at various levels, described in program terms.
- **Program—real-world (PRR):** Object descriptions using terminology which is valid in both real-world and program domains, and is abstract and shared across various problem domains.
- **Program—domain (PRD):** Object descriptions containing a mixture of program and problem domain references, or a reference which is equally valid in the program and problem domains.
- **Domain (DOM):** References to objects described in problem domain terms.
- **Indirect reference (IND):** An anaphoric reference to an object.
- **Unclear (UNO):** Object references that cannot be coded because they are ambiguous or unclear, or because the object which is being referred to cannot be identified.

Figure 2 gives an example of ODC coding.

Investigation

This section describes an investigation into inter-rater reliability of Good's scheme and the main reasons for differences among raters.

Method

Three raters learned to use Good's program summary analysis scheme and coded real program summary data. Problems in learning and using the scheme as well as differences between the raters were recorded and analyzed.

```
program task4 (input, output);
var weight,x,d,t: integer;
begin
     t := 0;
     write('Enter patient's weight (kg): '); readln(weight);
     for x := 1 to 3 do
          begin
          d := weight * 3;
          writeln('Day ', x, 'morning and evening ', d, 'ml.');
          t := t + 2*d
          end;
     for x := 4 to 7 do
          begin
          d := weight * 4;
          writeln('Day ', x, 'morning ', d, 'ml.');
          t := t + d
     end;
     writeln(t)
end.
```

Figure 3: English translation of the program summarized by students.

Raters: There were three raters, two male and one female, out of whom two were postgraduate students and one was a postdoc researcher.

Materials: Forty-four program summaries were gathered as part of another study (Kuittinen and Sajaniemi, 2003). They consisted of students' answers to a program comprehension question in an exam at an university level introductory Pascal programming course. Students' task was to "describe what is the purpose of the given program and how it works", i.e., to write a program summary. Figure 3 gives an English translation of the program; in the original program both output strings and variable names were Finnish words or abbreviations. All other variable names were single-letter meaningful abbreviations except the variable `weight` which was a problem domain word (`paino` in the Finnish version). In the English version, the variable `x` should be `d` for "day" but this would clash with `d` for "dose"; in the Finnish version this problem did not arise.

Eight program summaries were selected for training the raters and the remaining 36 summaries were used for the inter-rater validation. Table 1 gives statistics of the program summary sizes measured in words. (Due to the absence of articles and prepositions, and to the use of compound words[1] in Finnish, the same information contents is usually achieved with less words in Finnish than in English.) The training material was selected to include a diverse set of summary statements; therefore, they tended to be longer than summaries on average. Moreover, the longest summary was included in the training set because of its anomalous nature: it consisted mainly of program code fragments. Excluding this summary, the longest program summary in the training set was 276 words long.

Procedure: There was a training period for segmenting and coding; first for IT and then for ODC. The training consisted of explaining the classification instructions, segmenting the training material, comparing the results of segmenting in a meeting, and agreeing on the rules for the validation phase. During the validation all raters worked independently. Meetings were lead by a fourth person.

[1] In Finnish, words can be combined in the same style as "textbook" or "database" in English. However, in Finnish this is much more common than in English, e.g., "annosteluohje" for "dosage instructions". Because raters are not willing to split a word into two segments, we will use the notation "dosage-instructions" when such cases appear in our examples. As a compound word refers usually to a single object, this will probably not introduce language-dependent differences.

Problems were gathered by the raters, who made notes during training and validation, and by the meeting leader during meetings. Sources of problems and differences in the numeric results were discussed in several meetings after the validation phase.

Table 1: Program summaries. Lengths are measured as word counts.

	Training	Validation
Number of program summaries	8	36
Minimum length	114	42
Maximum length	349	285
Mean length	199.0	137.6
Std dev of length	74.6	57.5

Results

In addition to the IT categories in Good's scheme, we used the category **Continuation (CUT)** to cover cases where one segment is embedded in another segment. For example, in Figure 4 the information (lines 1 and 3) that the variable `t` is increased by a certain amount is interrupted by another information describing the variable (line 2). Hence, only the first line is given a proper IT category, and the third line is coded as a continuation to the first line.

The variable t,	Operations
which gives the total drug amount,	Data
is increased by the morning and evening doses.	Continuation

Figure 4: An information segment embedded in another segment. The latter part of the surrounding segment is coded as Continuation.

Table 2 gives the distribution of IT categories by each rater. The first part of the table gives mean percentages of the 36 program summaries for each IT category. The second part is obtained by grouping categories describing high-level program information, low-level program information, and other IT categories. Finally, the last row gives the proportion of high-level information when other, program-unrelated information is discarded.

The number of IT segments for each rater varied between 883 and 892. There were 811 segments that were common to all three raters, i.e., 8.6 % disagreement in segmenting. Out of these 811 common segments the three raters coded 65.5 % similarly and 94.0 % of the segments were given the same category by at least two raters. Looking at the coding of each of the raters separately, each rater agreed with at least one of the other two raters in from 84.1 to 85.2 % of the common segments.

Table 3 gives the distribution of ODC categories by each rater calculated as means of the percentages of the 36 program summaries.

Table 2: Percentages of IT categories by rater. HIG refers to high-level program information, LOW to low-level program information, and OTH to program-unrelated information. HIP is the proportion of high-level segments of all program-related segments.

		Rater					
		A		B		C	
Code	Information type	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev

FUN	Function	9.3	7.6	8.1	5.5	10.5	8.5
ACT	Actions	20.7	12.0	18.2	12.4	14.6	12.1
OPE	Operations	10.6	9.9	7.3	8.4	7.7	6.6
SHI	State-high	1.5	3.4	1.5	3.1	1.2	2.7
SLO	State-low	0.1	0.7	0.1	0.8	0.1	0.4
DAT	Data	24.5	11.1	27.4	12.2	29.9	15.4
CON	Control	4.9	5.3	3.8	4.8	4.8	5.3
ELA	Elaborate	21.6	14.9	26.3	17.1	20.4	16.6
MET	Meta	1.5	2.7	1.4	2.8	2.7	5.7
UNC	Unclear	0.0	0.0	0.8	3.4	0.8	3.0
INC	Incomplete	1.0	4.8	1.1	1.9	2.3	4.9
CUT	Continuation	4.2	4.1	3.9	4.5	5.0	4.9
HIG	FUN+ACT+SHI+DAT	56.0	18.2	55.2	20.6	56.2	20.2
LOW	OPE+SLO+CON	15.6	12.2	11.3	11.5	12.6	10.5
OTH	100-HIG-LOW	28.4	16.4	33.4	18.0	31.2	18.7
HIP	HIG/(HIG+LOW)*100	78.1	15.4	82.4	16.9	80.9	13.8

Table 3: Percentages of ODC categories by rater.

		Rater					
		A		B		C	
Code	Object Description Categ	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
PON	Program only	3.0	6.1	5.8	8.2	4.4	7.9
PRO	Program	15.8	15.1	22.5	16.3	20.5	16.6
PRR	Program—real-world	1.6	3.2	3.1	4.7	2.4	4.0
PRD	Program—domain	7.1	8.1	16.9	9.9	6.1	7.6
DOM	Domain	63.1	16.8	49.6	19.6	59.5	17.8
IND	Indirect reference	6.6	6.0	2.1	3.7	7.1	7.7
UNO	Unclear	2.9	6.5	0.1	0.8	0.0	0.0

The number of ODC segments for each rater varied between 712 and 734. There were 570 object descriptions that were common to all three raters, i.e., 21.0 % disagreement in segmenting. Out of these 570 common segments the three raters coded 73.2 % similarly and 99.3 % of the objects were given the same category by at least two raters. Looking at the coding of each of the raters separately, each rater agreed with at least one of the other two raters in from 81.1 to 98.8 % of the common segments.

Differences in Information Types

This subsection first lists reasons for inter-rater IT differences obtained by recording the problems experienced by the raters during the training, and by analyzing the differences observed in the validation. We start with problems in segmenting, i.e., what constitutes a segment, and continue with problems in selecting IT categories for segments. We conclude this subsection by looking at the frequencies of the problems in inter-rater differences.

Problems in Segmenting

The basic rule of segmenting program summaries to "short phrases consisting of a subject and a predicate (either of which may be implied)" (Good, 1999, p. 313) appears to be clear and unambiguous. However, the examples in (Good, 1999) and (Good and Brna, 2003) (which will be together called as *scheme defining documents* or SDD) contain exceptions to this rule, e.g., "by adding the adjustacent ones together" suggesting that non-finite clauses can be segments, and "[and then joins it to the other value it would have created if it had done what I just said] (complicated)" suggesting that even a single adjective, i.e., "complicated", referring to a different context than the rest of the sentence can be a segment. We found many occasions where the basic rule would lead to segments containing several information types described in detail below.

One might be tempted to suggest that several segments should be used only if the passage contains several IT categories. However, segmenting is supposed to be done first—independently of IT coding— which means that segmenting cannot depend on the number of IT categories present in the passage. Hence, the question of segmenting must be determined by the presence of *information items* independently of their types.

The "and" problem: The SDD contains contradicting examples of the effect of the word "and" on segmenting. The passage

> the program is selecting ... and allowing them to ...

is coded as a single segment (Good and Brna, 2003, p. 38) whereas

> It takes the numbers in the list and adds up numbers next to each other

is coded as two segments (Good, 1999, p. 320). However, in both examples, there is a predicate on both sides, and the subject is implied on the right-hand side. In our summaries, we found cases where both the subject and the predicate are implied:

> program tells how much and how many times a day

Contrasted with a slightly longer version:

> program tells how much and furthermore it tells how many times a day

it becomes unclear whether one should use one or two segments.

The non-finite clause problem: Non-finite clauses and other similar linguistic constructs give rise to passages that contain several information items even though there are no new subjects or predicates:

> once adjusted the numbers are added
>
> which tells using the value of x the number of the day

Special cases of this problem are specifications of time or other condition where the wording may contain a subject and a predicate or they may be absent:

> When we are at the beginning of the for-loop vs. At the beginning of the for-loop
>
> If the input is 0 [...] vs. With input 0 [...]

Time specifications can be even shorter and still carry an information item, e.g., "This time" or "Then", making it problematic to decide when a time specification should have its own segment.

Another special case of this problem is formed by *Meta* passages that may consist of a single word:

> presumably
>
> probably
>
> clearly

The information–dense passage problem: Some passages of text contain several information items and, indeed, several IT categories even though there is a single subject and predicate. For example, the following passage describes both data (the meaning of the variable `t`) and control (the assignment):

> The total drug amount is stored into the variable `t` at every round.

The example problem: Examples (that should be coded as *Elaborate*) may be very short or consist of tens of lines. The SDD notes that "(65 in this case)" is a single segment (Good and Brna, 2003, p. 39) but it is unclear how many segments should be made out of a long example. Furthermore, examples rarely have a subject or predicate, e.g.,

(kg)
i.e., single-dose,

Problems in Selecting IT Category

Coding problems considered either discrimination between two (or even three) IT categories, or were more general and not connected to any specific categories. Moreover, we found a need for two new IT categories. We start with the latter problem types.

The atmosphere problem: The SDD suggests (Good, 1999, p. 314) that coding can be carried out by category, i.e., by several passes through the summary in order to identify all segments of a particular type. In this style of coding, each segment is coded almost independently of the surrounding segments. However, a certain passage, e.g.,

Finally the amount of agent needed for doses is output.

may have a very different interpretation depending of its context. If this segment appears at the beginning of the summary where the function of the program is described, it represents *Function* information. If it appears at the end of summary as an explanation of the final output statement, it represents *Data*. Thus overlooking the context, or atmosphere, of the passage leads to incorrect coding.

The not-done problem: Program summaries contain passages that describe what the program does *not* do. The SDD does not explain how to code these. Examples are:

input cannot contain letters

and accepts even a negative value [i.e., there is no input check]

The missing "continuation" category problem: We found the need for a new IT category *Continuation* described at the beginning of this section.

The missing "irrelevant" category problem: We found the need for a new IT category *Irrelevant* to code information that is not related to the activity or results of the program, e.g.,

else the program works.

The name of the program is task4.

The program starts with the word " begin".

The program contains two for-loops.

Irrelevant information is usually valid—it just is not related to activities of the program or to its results.

The Operations vs. Actions problem: Activity within a loop that is described based on a single round (i.e., it should be coded as *Operations*) may appear to the reader as describing the effect of all rounds (i.e., an activity at the *Actions* level). The reader may even change the way the segment appears to him/her. Examples of this type are:

the variable t is increased again

The total drug amount is stored into the variable t at every round

The Data vs. Operations/Actions problem: There are many occasions where an information item may be interpreted to describe a variable (i.e., *Data*) or an activity (i.e., *Operations* or *Actions*). First, the specific words used to refer to an object may make a difference:

Finally the variable `t` is increased again	Operations/Actions ?
Finally the dose-total `t` is increased again	Data ?
Finally the gatherer `t` is increased again	Data ?

Even though these examples are very similar, the use of a natural language concept ("dose-total") or role name ("gatherer"[2]) seems to change the information from operation to data flow.

Second, individual verbs seem to be attached to different levels of abstraction even though the information content is basically the same, e.g.,

the size of the dose is assigned to the variable `d`	Operations
the size of the dose is computed into the variable `d`	Operations/Data ?
the variable `d` holds the size of the dose	Data

Third, it is not clear how to code an abstract activity (which should be coded as *Actions*) that refers at the same time to the total life-cycle of a variable (which should be coded as *Data*). A reference to the total life-cycle of a variable may even be indirect ("This time"), e.g.,

For the fourth day the dose changes	Actions/Data ?
The dose is determined by	Operations
This time the dose is determined by	Operations/Data ?

The Data vs. Control problem: Passages concerning control variables contain information about both control and the life-cycle of a variable:

where x steps from the value 1 to the value 3	Data/Control ?

The State-low vs. State-High problem: According to the SDD *State-high* relates to state described at an abstract level and *State-low* to state described at a low level (Good, 1999, p. 315). However, in the SDD examples *State-high* seems to be connected to loop termination conditions and *State-low* to `if`-conditions, independently of the level of the description, e.g.,

[program should terminate] when counter is greater than 4	State-high (Good, 1999, p.321)
if the head is greater than 180 ...	State-low (Good,1999, p. 196)

The Elaborate vs. some other category problem: Passages coded as *Elaborate* are examples or restatements of facts that have already been described. It is, however, unclear how far away the original fact is allowed to be, i.e., may the elaboration follow, say, 10 segments after the original fact, and how much can be deduced from earlier facts, i.e., may an elaboration contain a fact that is not explicitly said before but that can be easily deduced from earlier segments.

Furthermore, it is unclear whether descriptions of manner should always be coded as *Elaborate* as suggested by the examples in the SDD, e.g., (Good, 1999, p. 320):

It adds two successive numbers in the list	Function
putting a zero at the start	Elaborate

Rater Differences

The above list of problems in segmenting and coding was collected during the training and validation phases of the investigation. We devised *ad-hoc* solutions to problems that were found during training and the raters were advised to work accordingly. Nevertheless, there were differences between raters in the validation as described in subsection "Results".

Two problems accounted for more than half of the *differences in segmenting*: the *Meta* passages form of the Non-finite Clause Problem covered 27.7 % and the "And" Problem covered 23.8 % of the 231 non-common segments. Another 11.7 % of the non-common segments were caused by segments coded as *Continuation* by some rater(s) but not considered to cause segments by other(s). However, these are side effects of other differences in segmenting and do not explain anything by themselves.

Most interesting *differences in coding* the 811 common segments regard the 49 segments coded differently by all raters. Two cases accounted for a majority of these differences. The first was the Missing *Irrelevant* Category Problem that covered 30.6 % of the segment. The second was a

[2] Gatherer is one of the roles taught to the students in the original study. It can be compared to the concept of counter. While a counter counts something, a gatherer gathers the net effect of something, e.g., the sum of individual values.

combination of the *Operations* vs. *Actions* Problem and the *Data* vs. *Operations/Actions* Problem that covered 32.7 % of the segments. These segments were of the form

The total drug amount is stored into the variable `t` at every round

where a single assignment is *Operations*, the total effect of these assignments during all rounds of the loop is *Actions*, and the information that the variable `t` holds the total amount is *Data*.

Table 4 gives frequencies of information types in the 231 segments agreed by two raters but not by the third one. Taking the overall frequencies of the categories (Table 2) into account, the categories *Actions*, *Operations*, and *Incomplete* occur as codes for the problematic segments more often than expected, and the categories *Function*, *Data*, and *Continuation* occur less often than expected.

The most common problems were the *Data* vs. *Operations/Actions* Problem (22.5 %), the *Elaborate* vs. Some Other Category Problem (19.9 %), and the *Operations* vs. *Actions* Problem (11.7 %).

Table 4: Frequencies of information types in segments agreed by two raters only.

Information Type	Frequency
Function	7.6
Actions	25.3
Operations	12.4
State-high	0.0
State-low	0.0
Data	20.8
Control	4.6
Elaborate	19.5
Meta	1.5
Unclear	1.3
Incomplete	5.7
Continuation	1.2

Differences in Object Description Categories

We now turn to ODC inter-rater differences. We will first look at the problems in identifying objects, and then continue with problems in segmenting and coding followed by an analysis of differences observed in the validation.

Problems in Recognizing Objects

The data object definition problem: According to the SDD (Good, 1999, p. 316) ODC coding is applied to data objects; any other objects (e.g., the program, actions/events within the program, such as recursive call, iteration) should not be used for segmenting. During the training phase we looked at the differences between the raters and realized that the raters did not, however, agree on what objects should be used for segmenting. As a consequence, we decided that coding should be based on the following objects: weight, day, dose, total-amount, course-of-medication, and dosage-instructions. This decision was based on our idea of dividing objects into the following four categories:

proper data objects, e.g., total-amount, t

aggregate data objects, e.g., dosage-instructions, output

control objects, e.g., beginning-of-the-week, first for-loop

other/external, e.g., patient

The first example of each category above is a *Domain* description while the second is a *Program* description—for the fourth category no *Program* description is possible. The first two categories cover data objects and we selected them to be used for segmentation in the validation phase.

The synonym problem: Having decided the exact list of objects to look for, it was not always evident what words should be considered as synonyms for the selected objects. For example, under what conditions should the word "patient" be considered to be a synonym for the object "weight", or is the word "reading" a synonym for the object "dose" in the following:

morning and evening reading in milliliters

The natural object problem: As with the Synonym Problem, it is not always clear whether natural language data references should be understood as a representative of some object included in the analysis, e.g.,

plain number is multiplied by three

right amount of medicine

The "value" problem: The word "value" is an object by itself in all the examples of the SDD. However, in several cases we were unsure if this is really the case and could not found any grounds for considering "value" as an object, e.g.,

accepts a negative value

its value

that value

the value of the variable

Problems in Segmenting

Problems in deciding what constitutes a segment are of two types: whether a sequence of words should be split into several segments, and whether single-word references should be skipped under some circumstances totally.

The qualification problem: It is common to have an object qualified by another object. In the SDD, these result in a single segment except when the qualifier is the word "value", e.g., (Good, 1999, p. 323):

The value of

the element of heights is preserved with that iteration.

In some cases references to two objects seem to represent those objects themselves rather than the qualified unity. The following example set starts with this kind of situation and proceeds gradually to references where a single segment might be more appropriate:

dose of the third day

dose of the day [in the first round of the second loop]

dose of the day [in any round of the second loop]

dose of a day [some day]

dose-of-the-day

daily dose

These examples demonstrate that the wording itself does not always explain the perceived presence of one or two objects, but the context of the reference counts also.

The multi-word reference problem: Objects are often referred to by several words representing different ODCs leading to a need to segment each passage as a separate segment, e.g.,

total-amount (`m`)

`m`, the total-amount, ...

`m`, which is the total-amount, ...

`m`— `m` is the total-amount — ...

total-amount, which is kept in the variable `m`, ...

the total-amount contained in `m`

the total-amount is assigned to `m`

The variability is huge, and it is not at all clear how many segments should be used in each case. The category *Program—domain* may be used when there is a mixture of program and problem domain references (Good and Brna, 2003, p. 41), but the examples in the SDD, e.g., "a list of marks", are cases where both parts of the reference are compulsory to make the passage understandable. Furthermore, there are no special categories for other combinations.

The pronoun problem: In the SDD, all examples of *Indirect references* are personal pronouns. It is unclear whether other pronouns, e.g., "which" and "that", are anaphoric references.

The verbatim problem: Examples and code segments may contain verbatim references to objects, e.g.,

```
writeln('Day ', x, 'morning and evening ', d, 'ml.');
Then the dosage-instructions is output: "Day 1. morning and
evening 150 ml."
```

The expressions " `x`", " `d`", "Day", and "150 ml." do not actually refer to the objects but are verbatim copies of program or output text. It is unclear whether they should cause segmenting.

Problems in Selecting ODC Category

The context problem: This is similar to the Atmosphere Problem in IT coding but considers a single sentence rather than a larger context. In our material, sentences were much longer than those in the examples of the SDD, and the raters had different views of whether context should be taken into account.

The input coding problem: The SDD says that input is coded as *Program—real-world* but *Program* seemed to be more natural to the raters yielding differences between raters.

The Unclear-eagerness problem: The SDD does not state how eagerly the category *Unclear* should be applied: eager use will result in more reliable data but, of course, with fewer segments having a category that can be used in further analysis.

The natural language variable name problem: The name of one variable, `weight`, happened to be a natural language word meaning the purpose of the variable. This made it very hard to detect the correct category unless the context was obvious.

Rater Differences

Differences between raters were summarized in subsection "Results". Theoretically, there were *no differences in selecting the objects for segmenting* because we agreed on the set before the validation phase begun. However, *differences in segmenting* indicate that the recognition of even pre-defined objects is not easy: out of the 458 non-common segments, 63.6 % were due to the Synonym and Natural Object Problems. The Pronoun Problem covered 17.9 % of the non-common segments.

The vast majority of *differences in coding* were due to the Natural Language Variable Name Problem. Out of the 570 common segments, only four were coded differently by all raters and they all referred to the variable `weight`. Out of the 149 segments coded similarly by two raters, 79.9 % referred to this variable. Half of the rest were results of the Multi-word Reference Problem: the raters had used a single segment but based their coding on different aspects of the multi-word reference.

Discussion

Largest inter-rater differences in the IT category frequencies in Table 2 are over 5.0 % (*Actions* 6.1 %, *Elaborate* 5.9 %, and *Data* 5.4 %) but differences in the proportion of high-level information of program-related segments (HIP) were, however, smaller with the largest difference being 4.3 %. In practice (Corritore and Wiedenbeck, 1991; Good, 1999, Hoadley et al., 1996; Kuittinen and

Sajaniemi, 2003; Pennington, 1987), individual IT category frequencies are usually grouped to high-level and low-level information making the smaller variability in HIP important for research purposes.

Inter-rater differences in the ODC category frequencies (Table 3) were even larger: 13.5 % for *Domain*, 10.8 % for *Program—domain* , and 6.7 % for *Program*. The vast majority of the coding differences (80.4 %) were, however, caused by the poor selection of a natural language word for one of the variables in the program to be summarized. Segmenting was more problematic for ODC than for IT.

The problems listed in subsections "Differences in Information Types" and "Differences in Object Description Categories" are more important than their frequencies in explaining differences among raters: many problems did not manifest themselves as differences in the validation phase because we devised solutions to them during the training phase. Our solutions were, however, *ad-hoc* and cannot be considered as general solutions if the scheme is to be applied in more general settings. Table 5 lists the problems together with our suggestions for solution types: whether a revision of the scheme is required, whether the problem can be solved by more detailed documentation of the scheme, whether the problem is due to problems in programs to be summarized, or whether the problem is still open.

In general, the documentation of the scheme should be improved by increasing consistency of examples and by including the summarized programs in order to make the relationship between coding examples and the programs explicit. For example, in the segment (Good, 1999, p. 316):

and whether **the person** has passed	Domain

it is unclear whether persons are explicitly mentioned in the program (e.g., "Enter person's mark:") or only inferred (e.g., "Enter mark:"). This distinction is important in explaining the correct solution to the Synonym Problem.

IT problems that require changes in the scheme are the addition of two new categories, and the abandonment of the subject–predicate requirement. As the SDD already contains examples with no subject or predicate, the latter solution can be considered not to change the scheme but to be a documentation problem only. Another suggestion for improving the documentation is the use of the new concept *information item* to clarify the process of segmenting for IT analysis.

Other suggested IT documentation additions concern borderlines between individual categories, the atmosphere problem, and some special cases. The only open problem is the *Data* vs. *Operations/Actions* problem which covered 28.9 % of the differences among segments common to all raters and depends on delicate interpretation of the true meaning of object descriptions.

ODC problems requiring changes in the scheme concern the selection of objects and the detection of object references. The documentation should be more precise on these problems, and furthermore suggest that the objects are listed before segmenting the summaries—particular, if there are more than one rater. The use of the scheme could be extended to allow the use of a limited set objects, e.g., objects that are explicitly mentioned in the program text both with a domain name and with a program name in order to avoid the *Unclear*-eagerness Problem, or to allow the use of a limited set of references for segmentation in order to avoid the Synonym and Natural Object Problems.

Other ODC documentation problems concern the context problem, and some special cases. The Natural Language Variable Name Problem can be avoided by careful design of programs to be summarized. The scheme cannot solve this problem although the documentation should mention it.

		Problem	Scheme	Document.	Mater.	Open
IT	S	“and”		●		
		non-finite clause	●	●		
		information–dense passage	●	●		
		Example		●		
	C	Atmosphere		●		
		not-done		●		
		missing “continuation” categ.	●			
		missing “irrelevant” categ.	●			
		Operations vs. Actions		●		
		Data vs. Operations/Actions				●
		Data vs. Control		●		
		State-low vs. State-High		●		
		Elaborate vs. some other categ.		●		
ODC	O	data object definition	●	●		
		synonym		●		
		natural object	●	●		
		“value”	●			
	S	qualification	●	●		
		multi-word reference	●	●		
		pronoun		●		
		verbatim		●		
	C	context		●		
		input coding	●			
		Unclear-eagerness		●		
		natural language variable name		●	●	

Table 5: Suggested solution types for the problems. Scheme: revise the program summary scheme; Document.: revise the documentation of the scheme; Mater.: use appropriate experimental materials; Open: an open problem. The upper part contains problems related to IT, and the lower part problems related to ODC. S: segmenting problems; C: category selection problems; O: object recognition problems.

Conclusion

Program summary analysis can be used in computer science education and in research into the psychology of programming to study students' mental models and to assess the quality of comprehension. We have reported an investigation of the inter-rater reliability of Good's scheme (Good, 1999) for summary analysis.

The investigation consisted of a training phase and a validation phase. The differences in ratings were analyzed, and all problems encountered during the whole investigation were recorded, and possible solution types for the problems were discussed. The findings indicate that by improving the scheme and its documentation, most of the observed inter-rater differences can be avoided. The only open problem concerns making the distinction between descriptions of data and activities in cases where the specific words that are used, or the abstractness of expression may affect interpretation of the information type.

Acknowledgments

This work was partially supported by the Academy of Finland under grant number 206574.

References

Corritore C. L. and Wiedenbeck S. (1991) What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, 3(2):199–222.

Dillon A. and Gabbard R. (1998) Hypermedia as an educational technology: A review of the quantitative research literature on learner comprehension, control and style. *Review of Educational Research*, 68(3):322–349.

Good J. (1999) *Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension*. PhD thesis, University of Edinburgh.

Good J. and Brna P. (2003) Toward authentic measures of program comprehension. In *EASE and PPIG 2003, Papers from the Joint Conference at Keele University*, pages 29–49.

Hoadley C. M., Linn M. C., Mann L. M. and Clancy M. J. (1996) When, why and how do novice programmers reuse code? In Gray W. D. and Boehm-Davis D. A., editors, *Empirical Studies of Programmers: Sixth Workshop*, pages 109–129. Ablex Publishing Company.

Kuittinen M. and Sajaniemi J. (2003) First results of an experiment on using roles of variables in teaching. In *EASE and PPIG 2003, Papers from the Joint Conference at Keele University*, pages 347–357.

O'Brien M. P., Shaft T. M. and Buckley J. (2001) An open-source analysis schema for identifying software comprehension processes. In Kadoda G., editor, *Thirteenth Workshop of the Psychology of Programming Interest Group*, pages 129–146.

Pennington N. (1987) Comprehension strategies in programming. In Olson G. M., Sheppard S. and Soloway E., editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113. Ablex Publishing Company.

Sajaniemi J. and Kuittinen M. (in press) An experiment on using roles of variables in teaching introductory programming. *Computer Science Education.*

von Mayrhauser A. and Lang S. (1999) A coding scheme to support systematic analysis of software comprehension. *IEEE Transactions on Software Engineering*, 25(4):526–540.