Behaviour & Information Technology
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tbit20

A situated cognition view about the effects of planning and authorship on computer program debugging

Lai-Chong Law

Version of record first published: 08 Nov 2010.

To cite this article: Lai-Chong Law (1998): A situated cognition view about the effects of planning and authorship on computer program debugging, Behaviour & Information Technology, 17:6, 325-337

To link to this article: http://dx.doi.org/10.1080/014492998119283

BEHAVIOUR & INFORMATION TECHNOLOGY, 1998, VOL. 17, NO. 6, 325–337

A situated cognition view about the effects of planning and authorship on computer program debugging

LAI-CHONG LAW

University of Munich, Germany; email law@mip.paed.uni-muenchen.de

Abstract. Two experiments were conducted to investigate the relationship between planning and debugging and the effect of program authorship on debugging strategies. Three groups of participants with different programming experiences were recruited. In the first experiment, the participants were asked to develop and debug their self-generated program whereas in the second experiment, they were asked to debug an other-written program where some logical errors were planted. Situated cognition approach, being an emergent cognitive paradigm, furnishes an alternative framework to understand the problems of interest. Deweyan notion of inquiry and Gibsonian theory of affordance are of particular relevance. The results show that planning is ineffective for debugging, irrespective of the programming expertise level and program authorship. Besides, situated debugging is demonstrated to be the preferred strategy which is not significantly related to the program authorship. A model of planning for program debugging and a theory of two-faceted transparency are postulated for explicating the observations.

1. Introduction

As plans are broadly considered as one of the valuable resources for intellectual activities, a body of relevant research and literature can be located (e.g. Friedman *et al.* 1987, Allen *et al.* 1990). Whereas psychologists and artificial intelligence scientists tend to treat planning as an essential component of problem-solving which is grounded deeply in the information-processing paradigm (Newell and Simon 1972), philosophers tend to subsume the problem of planning under the concepts of rationality and intentionality (Bratman 1987). The early conceptions regarding the tripartite relationships between goals, plans, and actions have been laid down by the seminal works of Sacerdoti (1977) and Schank and Abelson (1977), upon which Soloway and his colleagues (1985, 1988) formulated their plan-based theory for programming expertise. Nonetheless, the classical assumption that plans play a strong controlling role in the actions they project has been challenged primarily because of its inadequacy to explicate how an agent copes with a complex and dynamic situation. Consequently, situated cognition (or variously known as *situated action*), arisen as a reaction to the inherent limitations of the traditional cognitive theories, has furnished an alternative theoretical framework for understanding the intricate relationships between plans and actions and instigated heated debates (Suchman 1987, 1993, Agre and Chapman 1990, Clancey 1993, Vera and Simon 1993a,b).

2. Philosophical roots of situated cognition theory

The basic tenet of situated cognition theory is that human cognition results from the interactions between intellectual agents and the sociohistorically constituted contexts in which they are embedded (Bredo 1994, Gruber *et al.* 1995). It emphatically advocates the interdependency between cognition and context, between knowledge and the community of practice, between problem solving and embodied activity, and between individual change and social change. Paradoxically, the key concept 'situated' is interpreted differently in accordance with the theoretical orientation of individual situationist (for details see Law 1997). For Suchman (1987), it essentially means 'embedded in the context of particular, concrete circumstances' and '*ad hoc*'. For Clancey (1993) it implies 'non-deliberative', 'ever-renewing' and 'non-mediating' (i.e. transcending symbolic representation). For Greeno (1992), qualifying cognition with the word situated is misleading as it may suggest the existence of its complement 'non-situated cognition'. Instead, he coined the term 'situativity' as a general characteristic of cognitive activities which should be understood

primarily as interactions between agents and physical systems and other people. In fact, the lack of a unitary definition is one of the drawbacks of this emergent paradigm which is therefore criticized as a loose school of thought. As the present study is basically inspired by Suchman's (1987) claims concerning the relationship between plans and actions, her interpretations of the concept situated is thus adopted.

2.1. *Deweyan perceptual and reflective inquiry*

Situated cognition is not a revolutionary idea, its roots trace back to different classical theories of which Dewey's (1938/1986) pragmatic social behaviourism plays a significant role (Garrison 1995). Among others, his notions of situation and inquiry are of particular significance (Burke 1994). In Dewey's terms, situations, occurring in ongoing activities of some given organism-environment system, are localized episodes of some sort of disequilibrium. Inquiry is conceived as an innate impulse of a living system to transform situations so as to counteract the instability arisen. An inquiry, which is depicted as progressive and circular, can be entirely perceptual, as if to proceed in a largely automatic manner, or it may also involve reflective processes. Basically, a perceptual inquiry entails adapting to perceived constraints of a situation which are cued by registered sensory data and results in ad-hoc actions (cf. Clancey's (1993) notion of coordination without deliberation), whereas a reflective inquiry entails eliciting propositions and examining cognitively possible consequences of each proposed act and leads to planned actions. Noteworthy is that an agent is not bound to a particular type of inquiry when tackling a specific problem, but rather shifts between the two types of inquiry. Indeed, the adoption of either perceptual or reflective inquiry at a certain moment is contingent on the ongoing interactions of various situational and personal cognitive-affective factors.

2.2. *Gibsonian theory of invariant and affordance*

In the same vein, situated cognition echoes Gibson's (1979/1986) notions of direct perception and affordance. In Gibson's views, organisms are adjusted or *attuned* to variables and invariants of information in their activities as they interact with other systems in the world. In this sense, perception is direct without entailing any mediating symbolic representation (cf. Dewey's (1938/1986) perceptual inquiry). It is reasonable to assume that invariants possess the dualistic property of being constant and flexible that allows experience to be gained and transported over time (Volpert 1985). Similarly, duality also holds for another key Gibson's concept—affordance—which is both physical and psychical. As a kind of resource, an affordance only exists under certain conditions of organism–environment interrelation, however, as it is a use-value, it does exist independently of being utilized (Reed 1987, Greeno 1994). In Gibson's (1979/1986, p. 129) words, an affordance is 'neither an objective property nor a subjective property; or it is both'. Indeed, the concept of affordance is undeniably problematic as shown by the contrasting interpretations ascribed to it (see e.g. von Hofsten 1985, Agre 1993, Vera and Simon 1993a). To delineate this elusive notion in a more concrete way, affordance can be defined as objects of the world which constrain and guide human actions. Whether agents can adapt their actions to the constraints and demanding features of the environment (i.e. *attunement to affordance*) depends essentially on their subjective perception of the functions of the objects which is in turn related to their sociocultural experiences. For instance, a book constrains a potential user to open and read it, that is, it is perceived as 'open-and-read-able'; however, it may also be used as a stand for supporting a person when it is perceived as 'stand-on-able'. The applicability of specific function relies on the goal of agent–world interaction. Nonetheless, owing to incompatible experiences, agents may fail to attune to the affordance of a particular situation. In summary, Gibsonian ecological psychology addresses the key question of what sources of information in the environment that organisms use in their activities.

3. Computer program debugging

Being deeply entrenched in the information-processing theory, planning is regarded as an essential component skill of computer programming. Computers, being a medium of pluralistic ideas, can be employed as an instrument for observing different planning and action styles of programmers (Turkle and Papert 1991). A key concept recurrently discussed in the research on the psychology of computer programming is *programming plan*, which is defined as a description of program fragments representing stereotypic action sequences in programming (i.e. template). Programming activities, including coding, comprehension and debugging, are found to be facilitated through plan retrieval and recognition. In particular, based on this concept, bug categorizations, debugging tools and tutoring systems for training novice programmers have been developed (e.g. Spohrer *et al.* 1985, Johnson 1990). One of the

major claims of the plan-based theory is that acquisition of programming expertise is solely determined by the possession of programming plans. However, it is this very claim that triggers debates. Diversified views have been put forward by researchers such as Davies (1991), Gilmore (1990), Green (1989), Rist (1989), etc. From the arguments posited by proponents and antagonists, the available data give rise to the impression that the plan-based theory has not grounded upon a firm base. Consequently, for future research, the emphasis should be shifted from studying static knowledge structures to dynamic applications of strategic knowledge.

4. Implications for the present study

Given that debugging is a highly dynamic activity, observations of the embodied skills of the participants when dealing with program errors, usually in reaction to diagnostic data, may shed some light in the understanding of the phenomenon of *breakdown* which is the primary concern of some situationists (e.g. Winograd and Flores 1986, Suchman 1987). Furthermore, in view of inadequate research on program debugging despite its significance (e.g. Vessey 1986, Gilmore 1991), situated cognition paradigm is expected to fuel some insights into the formulation of a comprehensive theory of debugging. Specifically, the exact relationship between planning and debugging, which has not been systematically studied, entails more research efforts. In addition, some early research on program debugging reports that programmers perform differently when debugging a self-generated program as compared with a program written by others (Gould 1975, Allwood and Bjöhar 1990). This problem can be re-considered from an innovative perspective. In summary, the two major aims of the present project are: first, to examine the role of planning in computer program debugging; second, to clarify the relationship between the program authorship and debugging strategies.

Based on the above elaborated situated action perspective, two debugging strategies—planned and situated—employed by programmers for coping with program errors are proposed. The former refers to the approach involving planned procedures whereas the latter denotes *ad hoc* actions. To operationalize the tendency of adopting either of the two debugging strategies, a measurement coined as 'plan-action-index' is devised (see below). Specifically, the following hypotheses are postulated which will be evaluated with the empirical results described subsequently. Hypotheses 1 and 2 will be examined in the first experiment and the other three hypotheses will be explored in the second experiment.

> *Hypothesis 1: Experts will demonstrate equivocal tendency to adopt either planned debugging or situated debugging strategy, whereas their less experienced counterparts will show a higher tendency to employ situated debugging strategy.*

A trend towards increasing plan use with expertise can be assumed, given that experts possess a large repertoire of knowledge from which programming plans (templates) can be retrieved and applied (Rist 1986). In addition, experts' well-developed embodied skills and situation-dependent thinking style allow them to respond situatively to the program bugs equally well (Vessey 1986). Hence, it is predicted that experts will use both strategies with more or less the same tendency. On the contrary, in view of their limited knowledge base, intermediates and novices are less adept to use plans. Nonetheless, their situation-independent thinking style and under-developed embodied skills may render their situated debugging ineffective.

> *Hypothesis 2: There will be significant negative correlations between the adjusted debugging time and the degree of planning for debugging a self-generated program.*

Given that one of the practical functions of a plan is to enhance the efficiency of problem solving by reducing the time required on the task, it is predicted that the higher the tendency to adopt planned debugging the shorter debugging time will be.

> *Hypothesis 3: Expert programmers, by virtue of their better program comprehension, which is essential for debugging other-written programs, will show a significantly higher tendency to plan than their less experienced counterparts.*

The suggestion that program comprehension is crucial for debugging has been verified empirically by different researchers (see, e.g., Gugerty and Olson 1986, Détienne 1990). Programming plans evoked in the comprehension process can be used as a kind of stencil with which the buggy code segment can be compared and the errors can thus be detected and corrected. The debugging strategy so elicited thus appears to be plan-based in nature.

> *Hypothesis 4: There will be significant negative correlations between the adjusted debugging time and the degree of planning for debugging an other-written program.*

The same arguments for Hypothesis 2.

Hypothesis 5: The tendency of planning will be significantly less for debugging an other-written than for debugging a self-generated program.

It is related to the problem of intention revelation which is presumably the key for program comprehension, which in turn is important for debugging. While the goal or intention of a self-generated program is intelligible to a programmer himself or herself, the goal of an other-written program tends to be incomprehensive.

5. Empirical design

The study consisted of Experiment 1 (Debugging a self-generated program) and Experiment 2 (Debugging an other-written program). On average, it took about 69.61 minutes (including coding time) and 28.06 minutes to finish Experiment 1 and Experiment 2, respectively. There was a break of about 10 minutes between the two experiments.

5.1. *Experiment 1: Debugging a self-generated program*

5.1.1. *Participants.* Thirty-five participants took part in this study. All of them were university students. They participated in the research project on a voluntary basis. Most of the experienced programmers enroled by responding to the announcement posted in the electronic bulletin board (newsgroups) whereas the novice programmers signed up in a lecture of an introductory course of computer science where the professor publicized the project. All the participants had programming experience in either Pascal or Modula2. The gender distribution of the male to female participants was 31:4. The average age was 24.4 years. Twenty-three of the participants were majoring in computer science, the others were taking computer science as a minor. As the number of semester was an unreliable and invalid indicator for programming expertise, two other criteria, namely the complexity of the program accomplished in the experiment and the self-reported programming experience (including formal training in secondary school as well as university, and self-learning), were taken into consideration for categorizing the participants into three respective groups. Three judges (the experimenter and two computer scientists) conducted independently *ex-post* classification work (cf. Vessey 1988). The average inter-reliability of the three judges was 87.5%. Consequently, the number of expert, intermediate and novice programmers were ten, thirteen, and twelve and their respective programming experiences were 9.4, 7.9 and 4.7 years. Nonetheless, one novice failed to finish the given task and his data were discarded.

5.1.2. *Instruments.* Various instruments were employed for the experiment:

(i) Computer hardware and software: A personal computer (IBM compatible) running under MS-DOS 5.0 and the 28-cm colour monitor, Turbo Pascal (Borland) DOS Version 6.0 Compiler, Gardens Point Modula-2 Compiler (1992), and MS-DOS-Editor (Version 1.1);
(ii) Video recorder: The experiment took place in the video-recording room at the Institute for Psychology of University of Munich. The camera focused primarily on the monitor, to record every single keystroke the participants invoked.

5.1.3. *Testing materials and procedures.* Some thinking-aloud exercises were conducted prior to the experiment proper in order to familiarize the participants with the skill required (Ericsson and Simon 1993). A problem for computing income taxes (Appendix A), which was devised by the experimenter herself, was evaluated to be of optimal difficulty and familiarity by the participants of different programming expertise in the pilot study. The participants were required to develop a program according to the given problem specifications, but *not* to compile the program until the first version was ready. After the participants reported to the experimenter that they found their program adequate, they were asked to compile it. After studying error messages or erroneous outputs, if any, they were required to explain the bugs and conceive a debugging plan, either verbally or in written form. However, they could initiate debugging on the computer instantly without making any plan or when they found their debugging plan adequate. The participants were asked to think aloud while debugging the program. They could refer to the notes made previously, if any. There was no time limit for all the tasks required. The whole process was videotaped and audiotaped.

5.1.4. *Data analyses.*
5.1.4.1. *Coding scheme for debugging activities.* A coding scheme is developed to analyse participants' think-aloud protocols generated during their debugging activities. Two major categories and their respective subcategories are depicted in Table 1a and 1b.

It was observed that reasoning about bugs or diagnostic information often concurs with debugging. Some situationists claim that reasoning occurs in sequences of behaviour over time, a kind of manifestation of perception-action coupling (Clancey 1993). The

fact that the 'planned debugging' and its actual implementation is very highly consistent shows that reasoning itself implies strongly the subsequent action.

5.1.4.2. *Computations of planning index and ad-hocecy index.* The Planning Index (*PI*) and the Ad-hocecy Index (*AhI*) are the measurements for indicating the tendency of a programmer to use planned strategy and situated strategy, respectively, for coping with program bugs. The total number of bugs generated varies substantially with individual participants. The larger the number of bugs, the more debugging behaviour will be required and consequently more debugging protocols will be generated. Hence, indices are equal to the ratio of the number of the protocols falling in a certain category to the total number of bugs. The formulae for computing the indices are shown Table 2. It is noteworthy that the indices can be greater than 1.0 when the subject employs different debugging strategies for dealing with the same bug. Specifically, the Plan-Action-Index for debugging Self-generating program (*PAI-SELF*), which is derived from the difference

Table 1a. Coding scheme for planned debugging

Code	Meaning	Definition
PD	Planning for Debugging	Specifying explicitly a sequence of operations for removing the bug detected.
EDP	Executing Debugging-Plan	Implementing the planned sequence of operations to remove the bug detected.
EIR	Executing Implicit-debugging-plan induced by Reasoning	Implementing the debugging processes implicitly referred to when the bug is reasoned.
BS	Bug-locating Strategy	Proposing the general strategy to locate the error.

Table 1b. Coding scheme for situated debugging

Code	Meaning	Definition
RB	Reasoning about Bug	Proposing some plausible causes for the bug detected by interpreting either the error message or the wrong output.
CRD	Concurrent Reasoning and Debugging	Proposing some plausible causes for the bug detected and simultaneously implementing the debugging processes implied by the proposed causes.
AhD	Ad-hoc Debugging	Correcting the bug almost instantly upon its detection, without any explicit or implicit pre-actional referral to the operations concerned.
RvD	Revising Debugging	Undoing the already executed debugging or re-correcting the error detected in an ad-hoc manner

Table 2. Formulae for computing Planning Index, Ad-hocey Index, and Plan-Action Index for Program Debugging

Index	Reference	Formula
PI[a]	Planning Index	$\frac{\text{Total No. of Planned-Debugging Protocols}}{\text{Total No. of Bugs}}$
AhI[a]	Ad-hocecy Index	$\frac{\text{Total No. of Situated-Debugging Protocols}}{\text{Total No. of Bugs}}$
PAI[b]	Plan-Action Index	$PI - AhI$

a In order to avoid confusion, for PI and AhI, subscript 1 and 2 are used to indicate 'Experiment 1' and 'Experiment 2' respectively.
b The extension -SELF and -OTHER are attached to differentiate self-generated program from other-written program.

between the Planning Index (PI_1) and the Ad-hocecy Index (AhI_1), functions as an indicator for inferring the debugging strategy of a programmer. A higher positive value indicates a greater tendency for resorting to planned debugging. Conversely, a larger negative value indicates a greater tendency for resorting to situated debugging.

5.1.4.3. *Adjustment of debugging time.* Obviously, the larger the number of bugs, the longer the debugging time. Hence, to evaluate the effect of planning on the efficiency of debugging, the debugging time is adjusted by dividing it by the total number of bugs.

5.1.5. *Results and discussion.*

5.1.5.1. *Evaluation of Hypothesis 1.* The values of PI_1, AhI_1 and *PAI-SELF* are summarized in Table 3. The hypothesis was confirmed by the observations that the intermediates and the novices demonstrated significantly higher tendency to employ situated debugging than planned debugging (within-group comparisons for each individual group, $t_{[12]}= 3.26$, $p< 0.01$; $t_{[11]}= 2.26$, $p< 0.05$) whereas the experts demonstrated more or less the same tendency in adopting either of the two debugging approaches. Further, the experts demonstrated a significantly higher tendency to adopt planned debugging than their less experienced counterparts ($F_{[2,31]}= 5.50$, $p< 0.01$). There was no between-group significant difference in the tendency to employ situated debugging. The difference between the frequency of using situated debugging and that of planned debugging was marginally significant (within-group comparisons for all participants, $t_{[33]}= 1.66$, $p< 0.1$), showing that situated debugging was a dominant approach.

5.1.5.2. *Evaluation of Hypothesis 2.* As consistent with the trend of the differences in the number of bugs, there were significant between-group differences in the absolute debugging time ($F_{[2,31]} = 15.25$, $p < 0.001$) with the experts' being the shortest and the novices' the longest (see Table 4). However, contrary to the expectation, the correlation between *PAI-SELF* and the adjusted debugging time was insignificant for all subjects. The results suggested that the planned debugging was not efficient enough to foster program debugging.

5.2. *Experiment 2: debugging an other-written program*

5.2.1. *Participants.* As Experiment 1. However, three novices failed to finish the task and their data were discarded.

5.2.2. *Instruments.* As Experiment 1.

5.2.3. *Testing materials and procedures.* A program, in which some plan-based or logical bugs were planted, was an adapted version of Johnson's (1990) 'Rainfall Problem' (Appendix B). The evaluations in the pilot study showed that the difficulty and familiarity level of the problem were acceptable. The participants were presented with a hard copy of the buggy program and asked to conceive a debugging plan offline, either verbally or in a written form, when comprehending the program. However, they could initiate compiling and debugging online without making any plan or when they found their plan adequate. Subsequently, if there were error messages or erroneous outputs, they were required to explain the bugs and conceive a debugging plan, but they could also initiate debugging on the computer instantly without making any further plan or when they found their plan sufficient. The participants were asked to think aloud while debugging the program. They could refer to the notes made previously, if any. There was no time limit for all the tasks required. The whole process was videotaped and audiotaped.

5.2.4. *Data analyses.* The same schemes for Experiment 1 (Table 1a and 1b) were employed for the present task. The Plan-Action-Index for debugging an other-written program (*PAI-OTHER*), which sum-

Table 4. Results of counts of bugs and debugging time for the self-generated program.

Level of expertise	n	Number of bugs	Debugging time (minutes)
Novice	11	16.45 (6.90)	48.94 (23.49)
Intermediate	13	15.15 (6.20)	34.52 (12.87)
Expert	10	5.50 (2.64)	32.29 (21.57)

Table 3. Results of the three indices for debugging the self-generated program.

Level of expertise	n	Mean Planning Index [PI_1]	Mean Ad-hocecy Index [AhI_1]	Mean *PAI-SELF*
Novice	11	0.40 (0.16)	0.54 (0.15)	– 0.13 (0.20)
Intermediate	13	0.36 (0.17)	0.71 (0.24)	– 0.36 (0.40)
Expert	10	0.65 (0.31)	0.51 (0.42)	0.14 (0.70)

marizes the tendency to adopt planned debugging and situated debugging strategies, was derived from the difference between the Planning Index (PI_2) and the Ad-hocecy Index (AhI_2) (see Table 2). The debugging time was adjusted by dividing it by the total number of bugs, including those planted by the experimenter and those introduced by the over-corrections of the participants.

5.2.5. *Results and discussion.*

5.2.5.1. *Evaluation of Hypothesis 3.* The hypothesis was refuted by the non-significant between-group differences in PI_2. There were non-significant between-group differences in AhI_2 and the resultant Plan-Action-Index (*PAI-OTHER*) (Table 5). This could be explained by the fact that, since the experts had already located most of the bugs in the offline planning phase, for the remaining bugs which emerged when the program was executed or the newly introduced bugs created by over-corrections, the experts debugged situatively because they understood the mechanism of the program. In contrast, the less experienced programmers resorted to tinkering (*ad hoc* trial-and-error approach) when they had difficulty in comprehending the program and diagnostic data, though they might attempt to use problem-solving strategies in the first place. Such tinkering usually introduced more bugs rather than eliminated the original planted ones. This supposition could be verified by the significant between-group differences in the total number of bugs fixed ($F_{[2,28]}= 6.09$, $p< 0.01$).

5.2.5.2. *Evaluation of Hypothesis 4.* There were significant between-group differences in the debugging time ($F_{[2,28]}= 4.44$, $p< 0.05$) with the experts' times being the shortest, novices' times the longest and the intermediates' times between the two (see Table 6). The hypothesis was rejected by the insignificant correlation between the plan-action-index (*PAI-OTHER*) and the adjusted debugging time. Being consistent with the results of debugging self-generated programs (see section 5.1.5.), but contradictory to the empirical findings of previous research (e.g. Vessey 1986), the alleged effect of planning in debugging by reducing the time required to attain the goal could not be verified by the present study.

5.2.5.3. *Evaluation of Hypothesis 5.* The hypothesis was refuted. There were no significant differences between the two indices. Of the three groups, the intermediates consistently demonstrated their hacker-approach to the task at hand with both *PAI-SELF* and *PAI-OTHER* being negative, that is, they did not tend to adopt planning but favoured situated actions. Comparatively, the experts showed their planner approach and the novices were somewhat ambiguous in adopting a particular stance. Furthermore, no significant within-group differences between the two indices could be found, suggesting that generally the subjects did not vary their planning behaviour substantially with the task at hand.

6. General discussions

6.1. *A model of planning for program debugging*

Theoretically, the deployment of situated actions for a task depends much on an agent's ability to appreciate the use of the very situation in which the task is embedded (cf. Dewey's (1938/1986) perceptual inquiry and Gibson's (1979/1986) direct perception; see also section 2.1. and 2.2.). In the case of debugging, the success of the task is determined by a programmer's understanding of diagnostic information, including error messages, wrong outputs, and malfunctionings of a program. Experienced programmers are capable of understanding diagnostic information because of their

Table 6. Results of counts of bugs and debugging time for the other-generated program.

Level of Expertise	n	Number of bugs	Debugging time (minutes)
Novice	9	10.78 (1.48)	20.56 (7.12)
Intermediate	13	12.42 (3.45)	19.59 (14.19)
Expert	10	8.80 (1.32)	8.78 (3.14)

Table 5. Results of the three indices for the other-written program.

Level of expertise	n	Mean Planning Index [PI_2]	Mean Ad-hocecy Index [AhI_2]	Mean *PAI-OTHER*
Novice	9	0.51 (0.14)	0.74 (0.18)	–0.23 (0.23)
Intermediate	12	0.44 (0.25)	0.72 (0.31)	–0.27 (0.51)
Expert	10	0.59 (0.29)	0.70 (0.23)	–0.10 (0.42)

training and experience, so they are enabled to deal with the debugging task situatively. By asking experienced programmers to devise an action plan, we see that they can transform the otherwise *ad hoc*, apparently automatic actions, into a coherrent plan (cf. Dewey's (1938/1986) reflective inquiry). In contrast, owing to their undeveloped skills, less experienced programmers fail to understand diagnostic information and instead employ some problem-solving strategies. First, they may attempt to interpret the diagnostic information in their own idiosyncratic ways or resort to some para-debugging means (e.g. simulation run, seeking help from other resources). Next, based on such reasoning about the bug, they formulate some sort of debugging plan whose execution may lead to bug elimination or over-correction, if the interpretation of the diagnostic information is originally erroneous. Nonetheless, some may eschew the reasoning process which is regarded as too cognitive-demanding. Consequently, they may resort to tinkering by attempting (cf. VanLehn's (1990) Repair Theory) some harmless, irrelevant or cosmetic changes which, in terms of their instant occurrence, appear *ad hoc* to an observer. These changes usually result in some over-corrections. The foregoing arguments are schematized in Figure 1.

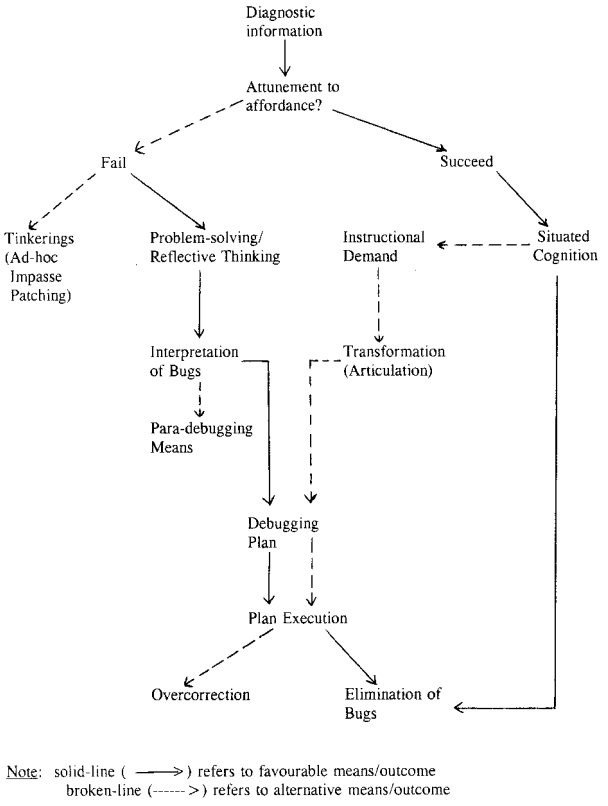

Figure 1. A model of planning for program debugging.

6.2. *Debugging strategies and authorship of a program: a theory of two-faceted transparency*

A program's authorship constrains debugging strategies because of its opaqueness of underlying intention. Intention-in-action is a psychological construct commonly employed by people to make sense or reason about others' visible conducts (Searle 1983, Bratman 1987). When the intelligibility of a representational system is beyond recognition, interactions with such a system are rendered extremely difficult (cf. Suchman's (1987) notion of mutual intelligibility). Consequently, in order to establish a cooperative relation with an 'alien' system, thereby successful interaction with its conceptual elements can be possible, *de-opaqueness* of such a system via some form of reasoning is deemed necessary. This process of *intent revelation* (or *reversed engineering*) can also be seen as the agent's attempt to own the system, so that he can define his identity in relation to the system and simultaneously gain a sense of control which has motivational effect in fostering the agent to deal with the system. From the situated view of learning, owning a problem is one of the determining factors for effective learning. Paradoxically, the full transparency of one's own actions somehow does not enable one to disengage from the current situation to reflect about the nature of the practice at hand and the functions of the attendant physical and conceptual objects which one tends to take for granted. Theoretically, a self-instituted system may become non-transparent in case of breakdown when action does not accomplish the goal effortlessly, that is, action can no longer be automatic but necessitates some sort of conscious deliberations. Presumably, two types of transparency are involved in the relation between program debugging and the authorship of the program: *transparency of intent* and *transparency of action*. Whereas the former can facilitate debugging, the latter somehow impedes debugging. The full integration between an agent and the environment, which depends much on the skill of the agent, is less likely to occur for novices than experts, and therefore the coincidence of the two aspects of transparency is defective for novices. In fact, inadequacy of domain-specific knowledge can give rise to a kind of breakdown which entails deliberative thinking with rules and representations accountable for actions coming into play.

In the case of debugging a self-generated program, the two aspects are segregated with transparency of intent still being retained but transparency of action being transformed to a state of (partial) opaqueness. Such an opaqueness of action forms the (bounded) practical context for current thinking. This is the very concept of breakdown rooted in Heidegger (1962, quoted by

Winograd and Flores (1986)). Breakdown in the present context refers to the emergence of program bugs which upset the agent-world equilibrium and impede the smooth execution of actions (cf. Dewey's (1938/1986) interpretation of situation). In order to restore the blocked cycle of activity, the agent tends to engage in conscious problem-solving by elaborating the information about the situation where the blockage occurs, at least down to the level where diagnostic cues and repair activities can be represented. Prior to resorting to reflective thinking, presumably the agent first attempts (probably unconsciously) to use perceptual ability to resolve the dilemma, however unreliable such an ability can be, especially for novices. The transparency of intent may lead to the problem of *einstellung effect* or fixedness (Allwood and Björhag 1990), when the error is a mistake rather than a slip (Norman 1981, Reason 1987). Slips occur when a person's actions are not in accordance with the actions actually intended (i.e. a good plan with poor execution), whereas mistakes are actions performed as intended but with effects turning out not to be in accordance with the intended goal (i.e. a deficient plan). Cognitive conflicts created by situational feedbacks, like diagnostic information, may invoke reconsiderations of an intention or a plan.

On the contrary, in case of dealing with other-written programs, both two aspects of transparency are detached and alien to an agent, and therefore the agent-environment interactions are not supported. In order to interact with the materials in the context, the agent has to address the problems pertinent to the transparency of intent which may eventually lead to the transparency of action, depending on whether the agent is sufficiently persistent in carrying through debugging actions to their conclusions. As intent revelation plays the pivotal role in the task of debugging an other-written program, the concomitant question about how the agent attains the level of intentional transparency is a primary concern. The mechanisms involved can be depicted as a cyclical process entailing *ongoing dialogues* with the conceptual elements embedded in the environment (Figure 2).

First, the agent attempts to anchor her thinking in the environmental salient features such as the given problem specifications and the syntactical structures on the hard copy of the program. This initial attempt of establishing a preliminary form of interactive relation between the agent and the immediate environment, if successful, will lead to an *action schema* which is rooted in relevant past experiences. The agent adapts the intention underlying the triggered action schema to the current situation so as to formulate a kind of *tentative intent*. Next, the agent continues her efforts in constructing some meaningful relationships between the conceptual elements of the

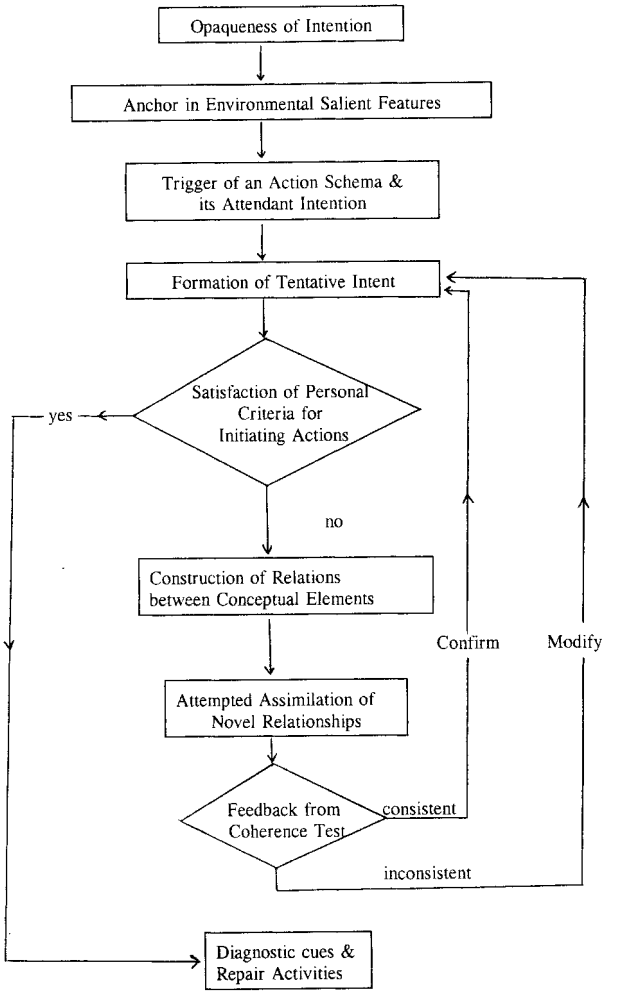

Figure 2. Mechanism of intent revelation.

programming language. In fact, such a relation-building activity can be seen as the part and parcel of program comprehension. Assuming the validity of the tentative intent, the agent tries to assimilate the newly constructed relationships into this existing scheme. Feedbacks from the *coherence test*, which is actually some sort of mental simulation of incorporating the novel relationships into the current scheme, inform the agent whether the provisional intent should be modified or not. Finally, if the agent finds that the extent of intent revelation, which is not necessarily complete, is sufficient for action initiation, then debugging activities will commence. As remarked previously, without a minimal level of transparency of intention, meaningful actions cannot proceed. The process of intent revelation is not only restricted to the stage of program comprehension, but rather is continuous up to the point when an error-free program is available. Presumably, prior to the next

occurrence of breakdown (i.e. emergence of bug), the intent and the action implied by the existing interactional model is transparent to the agent.

In view of the involved mechanism of intent revelation, it is reasonable to conclude that debugging an other-written program is practically much more challenging than debugging a self-generated program when the complexity of the programs is comparable.

7. Conclusions

Planning for program debugging is demonstrated to be ineffective, irrespective of program authorship and programming expertise. This may be attributed to the fact that debugging is such a highly complex and dynamic activity that it is not amenable to planning, but rather entails situated actions. Besides, situated actions are demonstrated to be the preferred debugging strategy which is not significantly related to the program authorship. Conclusively, a cluster of factors relating to the planning behaviour of a programmer can be enumerated:

(i) the nature of the task, especially its complexity and novelty;
(ii) the predictability of the concomitant changes in the related situational artifacts;
(iii) the resourcefulness of the environment which affords the agent-world interactions;
(iv) the size of the repertoire of programming plans from which relevant schemata will be triggered when the agent actively interacts with the environment;
(v) the range of skills which enable understanding of a specific situation in which the task is embedded, and enhance the adaptation of the triggered action schema to the specific situational constraints;
(vi) the motivational and emotional stance, which influences the tendency to satisfy the instructional demand and other situational needs;
(vii) personal characterisitcs of the programmer, including impulsiveness and preferences for planning.

Consequently, debugging expertise should not be defined merely in terms of knowledge structures, but rather programming culture constituted by notations of a programming language, programming discourse rules, human–computer interactions, and tutor–learner as well as peer–programmer interactions. Cognitive apprenticeship approach (Collins *et al.* 1989), a situated learning model grounded upon social constructivist theories of Vygotsky (1978, 1986), endorses the assumption that the processes of exposing fallacies in practical reasoning and activating inert knowledge are amenable to social interactions and thus ascribes much importance to expert-modelling and collaborative learning. This innovative approach appears to be an appropriate instructional model for training debugging skills. Nonetheless, the validity of such a supposition is another empirical query.

References

Agre, P.E. 1993, The symbolic worldview: Reply to Vera and Simon, *Cognitive Science*, **17,** 61–69.

Agre, P.E. and Chapman, D. 1990, What are plans for? in P. Maes (ed.), *Designing autonomous agents: Theory and practice from biology to engineering and back* (Cambridge, MA: MIT Press), pp. 17–34.

Allen, J., Hendler, J. and Tate, A. (eds) 1990, *Readings in planning* (San Mateo, CA: Morgan Kaufmann Publishers).

Allwood, C.M. and Björhag, C.-G. 1990, Novices' debugging when programming in Pascal, *International Journal of Man–Machine Studies*, **33,** 707–724.

Bratman, M.E. 1987, *Intention, plans, and practical reason* (Cambridge, MA: Harvard University Press).

Bredo, E. 1994, Reconstructing educational psychology: Situated cognition and Deweyian pragmatism, *Educational Psychology*, **29** (1), 23–35.

Burke, T. 1994, *Dewey's new logic: A reply to Russell* (Chicago: University of Chicago Press).

Clancey, W.J. 1993, Situated action: A neuropsychological interpretation response to Vera and Simon. *Cognitive Science*, **17,** 87–116.

Collins, A., Brown, J.S. and Newman, S.E. 1989, Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics, in L.B. Resnick (ed.), *Knowing, learning and instruction: Essays in honour of Robert Glaser* (Hillsdale, NJ: Lawrence Erlbaum Associates), pp. 453–494.

Davies, S.P. 1991, The role of notation and knowledge representation in the determination of programming strategy: A framework for integrating models of programming behaviour, *Cognitive Science*, **15,** 547–572.

Détienne, F. 1990, Expert programming knowledge: a schema-based approach, in J.-M. Hoc, T.R.G. Green, R. Samurçay and D.J. Gilmore (eds), *Psychology of programming* (London: Academic Press), pp. 205–222.

Dewey, J. 1938 [1986], *Logic: The theory of inquiry,* in J.A. Boydston (ed.), *John Dewey: The later works, 1925-1953, Vol.12* (Carbondale and Edwardsville: Southern Illinois University Press), pp. 1-5-27.

Ericsson, K.A. and Simon, H.A. 1993, *Protocol analysis: Verbal reports as data*, rev. edn (Cambridge, MA: MIT Press).

Friedman, S.L., Scholnick, E.K. and Cocking, R.R. 1987, *Blueprints for thinking: The role of planning in cognitive development* (Cambridge: Cambridge University Press).

Garrison, J. 1995, Deweyan pragmatism and the epistemology of contemporary social constructivism, *American Educational Research Journal*, **32** (4), 716–740.

Gibson, J.J. 1986 [1979], *The ecological approach to visual perception* (Hillsdale, NJ: Erlbaum).

Gilmore, D.J. 1990, Expert programming knowledge: a strategic approach, in J.-M. Hoc, T. R.G, Green, R. Samurçay and D.J. Gilmore (eds), *Psychology of programming* (London: Academic Press), pp. 223–234.

Gilmore, D.J. 1991, Models of debugging, *Acta Psychologica*, **78,** 151–172.

Gould, J.D. 1975, Some psychological evidence on how people debug computer programs, *International Journal of Man–Machine Studies*, **7,** 151–182.

Green, T.R.G. 1989, Cognitive dimensions of notations, in A. Sutcliffe and L. Macaulay (eds), *People and computers V* (Cambridge: Cambridge University Press), pp. 443–460.

Greeno, J.G. 1992, The situation in cognitive theory: Some methodological implications of situativity. Paper presented at the American Psychological Society (June: San Diego, CA).

Greeno, J.G. 1994, Gibson's affordance, *Psychological Review*, **101** (2), 336–342.

Gruber, H., Law, L.-C., Mandl, H. and Renkl, A. 1995, Situated learning and transfer: State of the art, in P. Reimann and H. Spada (eds), *Learning in humans and machines. Towards an interdisciplinary learning science* (Oxford: Pergamon), pp. 168–188.

Gugerty, L and Olson, G.M. 1986, Comprehension differences in debugging by skilled and novice programmers, in E. Soloway and S. Iyengar (eds), *Empirical studies of programmers* (Norword, NJ: Ablex), pp. 13–27.

Johnson, W.L. 1990, Understanding and debugging novice programs, *Artificial Intelligence*, **42,** 51–97.

Law, L.-C. 1997, The role of plans and planning in the development of computer programming expertise: A situated action view. Doctoral dissertation, Institut für Empirisch Pädagogik und Pädagogische Psychologie, Ludwig-Maximilians-Universität München.

Newell, A. and Simon, H.A. 1972, *Human problem solving* (Englewood Cliffs, NJ: Prentice Hall).

Norman, D.A. 1981, Categorization of action slips, *Psychological Review*, **88** (1), 1–15.

Reason, J. 1987, The psychology of mistakes: A brief review of planning failures, in J. Rasmussen, K. Duncan and J. Leplat (eds), *New technology and human error* (Chichester: John Wiley & Sons), pp. 45–52

Reed, E. 1987, Why do things look as they do? The implications of James Gibson's The ecological approach to visual perception, in A. Still and A. Costall (eds), *Cognitive psychology in question* (Sussex: Harvester Press), pp. 90–114.

Rist, R.S. 1986, Plans in programming: Definition, demonstration, and development, in E. Soloway and S. Iyengar (eds), *Empirical studies of programmers* (Norword, NJ: Ablex), pp. 28–47.

Rist, R.S. 1989, Schema creation in programming, *Cognitive Science*, **13,** 389–414.

Sacerdoti E.D. 1977, *A structure for plans and behaviour* (North Holland, New York: Elsevier).

Schank, R.C. and Abelson, R. 1977, *Scripts, plans, goals, and understanding* (Hillsdale, NJ: Erlbaum).

Searle, J.R. 1983, *Intentionality: An essay in the philosophy of mind* (Cambridge: Cambridge University Press).

Soloway, E.,1985, From problems to program via plans: The content and structure of knowledge for introductory LISP programming, *Journal of Educational Computing Research*, **1,** 157–172.

Soloway, E., Adelson, B. and Ehrlich, K. 1988, Knowledge and processes in the comprehension of computer programs, in M.T.H. Chi, R. Glaser and M.J. Farr (eds), *The nature of expertise* (Hillsdale, NJ: Lawrence Erlbaum Associates), pp. 129–152.

Spohrer, J., Soloway, E. and Pope, E. 1985, A goal-plan analysis of buggy Pascal programs, *Human–Computer Interaction*, **1,** 163–207.

Suchman, L. 1987, *Plans and situated actions: The problem of human machine communication* (Cambridge: Cambridge University Press).

Suchman, L. 1993, Response to Vera and Simon's situated action: A symbolic interpretation, *Cognitive Science*, **17,** 71–75.

Turkle, S. and Papert, S. 1991, Epistemological pluralism and the revaluation of the concrete, in I. Harel and S. Papert (eds), *Constructionism* (Norwood, NJ: Ablex), pp. 161–191.

VanLehn, K. 1990, *Mind bugs: The origins of procedural misconceptions* (Cambridge, MA: MIT Press).

Vera, A.H. and Simon, H.A. 1993a, Situated action: A symbolic interpretation, *Cognitive Science*, **17,** 7–48.

Vera, A.H. and Simon, H.A. 1993b, Situated action: Reply to reviewers, *Cognitive Science*, **17,** 77–86.

Vessey, I. 1986, Expertise in debugging computer programs: An analysis of the content of verbal protocols, *IEEE Transactions on Systems, Man and Cybernetics*, SMC-16, **5,** 621–637.

Vessey, I. 1988, Expert-novice knowledge organization: An empirical investigation using computer program recall, *Behaviour and Information Technology*, **7** (2), 153–171.

Volpert, W. 1985, Epilogue, in M. Frese and J. Sabini (eds), *Goal directed behavior: The concept of action in psychology* (Hillsdale, NJ: Erlbaum), pp. 357–365.

von Hofsten, C. 1985, Perception and action, in M. Frese and J. Sabini (eds), *Goal directed behavior: The concept of action in psychology* (Hillsdale, NJ: Erlbaum), pp. 80–96.

Vygotsky, L.S. 1978, *Mind in society: The development of higher psychological processes* (Cambridge: Harvard University Press).

Vygotsky, L.S. 1986, *Thinking and language* (Cambridge: MIT Press).

Winograd, T. and Flores, F. 1986, *Understanding computers and cognition: A new foundation for design* (Norwood, NJ: Ablex).

Appendix A: *Problem Specification — Tax Problem**

The rules of taxation are described as follows:

Taxation	Married	Single
The first $10000 of the yearly income	20%	15%
The second $10000 of the yearly income	25%	20%
The third $10000 of the yearly income	30%	25%
The rest	35%	30%

In addition, there is a maximum taxation: 32% and 28% of the yearly income for a married couple and a single person, respectively. In other words, the total amount of income tax should not exceed this limit.

Write a program for computing income taxes according to the above rules. The data are inputted via the keyboard and should consist of an identity number, a 'M' or 'S' (where 'M' means married and 'S' single), and the yearly income. The program should process the input and generate the following output:

1) Identity number (ID no.)
2) Marital status (married or single)
3) Yearly income
4) Income tax
5) Net income

Check your program with the following data:

ID no.	Marital status	Yearly income
1001	M	12364.80
1002	S	8941.00
1003	M	112000.00
1004	S	26875.70
1005	S	15300.50
1006	M	34320.60

Sample output:

1001	married	12364.80	2591.20	9773.60
1002	single	8941.00	1341.15	7599.85
1003	married	112000.00	35840.00	76160.00

Suggestion: Procedure(s) can be used for performing the calculations required, but it is not absolutely necessary.

* The original problem specifications were presented in German.

Appendix B *Modifications of the original Rainfall Problem*

```
program Rainfall;
var
  Rain, Totalrain, Highrain, Averain: real;
  Days, Raindays, Drydays:integer;  <- - - inserted line (enhancing comprehensibility)
begin
  Rain:= 0;
  Totalrain := - 99999.0;           <- - - - inserted bug (violation of discourse rule)
  Days := - 1;                      <- - - - inserted bug (violation of discourse rule)
  repeat
    write('Enter rainfall : ');
    readln(Rain);                   <- - - - merged line (increasing brevity)
    writeln;                        <- - - - inserted line (enhancing comprehensibility)
    while Rain < 0.0 do
      begin
        writeln(Rain:0:2, ' is not possible, try again! ');
        write('Enter rainfall : ');  <- - - - inserted line (enhancing comprehensibility)
        readln(Rain);                <- - - - merged line (increasing brevity)
        writeln;                     <- - - - inserted line (enhancing comprehensibility)
      end;
  Days := Days + 1;                  <- - - - relocated line (planting a logical bug)
  Total rain := Totalrain + Rain;    <- - - - relocated line (planting a logical bug)
  while Rain <> 99999 do
    begin
      if Rain > 0 then
        Raindays := Raindays + 1;
      if Highrain >= Rain then       <- - - - amended line (planting a logical bug)
        Highrain := Rain;
    halt;                            <- - - - inserted line (avoiding computer hang-up)
    end;
  until Rain= 99999;
  Averain := Totalrain/Days;
  writeln(Days, ' valid rainfalls were entered');
  writeln('The average rainfall was ',Averain:0:2, ' mm');
  writeln('The highest rainfall was ', Highrain:0:2, ' mm');
  writeln('There were ', Raindays, ' rainy days in this period');
  writeln('There were ', Drydays, ' dry days in this period');
                                     <- - - - inserted line (a logical bug)
end.
```

Note: The underlined lines are the modifications introduced to the original program adopted by Johnson (1990) for his research project. Comments are presented on the right hand side to indicate the nature of the change as well as its purpose (enclosed within the brackets).