

# An Empirical Study of Programmer Learning during Incremental Software Development

Shaochun Xu, Václav Rajlich, and Andrian Marcus  
*Department of Computer Science*  
*Wayne State University*  
*Detroit, Michigan USA 48202*  
*{shaochun, rajlich, amarcus}@wayne.edu*

## Abstract

*The paper presents a case study that investigates programmer learning during incremental program development. Dialog based protocol and protocol analysis are used to explore the learning from the point of view of self-directed learning theory. The replicated case study confirmed that the Bloom's levels of cognitive domain and the cognitive activities of absorption, denial, reorganization, and expulsion characterize the programmer learning and give insight into the differences between experts and intermediate level programmers.*

## 1. Introduction

Software development is a knowledge intensive activity that involves not only programming, but also gathering relevant knowledge and encoding it in the software. Therefore, learning is a substantial component of software development. However, little research has been conducted in terms of programmer learning and related cognitive activities during software development. Curtis [7] indicated that the development of large software systems is composed of learning activities such as, communication and negotiation. Fisher et al. [10] proposed a support for the incremental development based on a specific knowledge model, where seeding, evolution and reseeding are the three stages of knowledge capture and transformation. Henninger [15] recognized that software development is a process involving various knowledge resources, which keep changing during the process. In our earlier work [26, 27], we introduced a self-directed learning model and applied it to incremental software development.

Since our earlier study was based on other people's data that were published in [19], in this paper we present a replicated study with the goal of answering the following questions:

- Can the Martin and Koss' case study [19] be replicated with other programming pairs?
- Do different pairs follow the same cognitive process?

- If different pairs follow different cognitive processes, what are the differences?
- Are there differences between intermediate level programmers and experts?

The replicated case study is based on pair programming, test-first, and refactoring practices. Pair programming refers to the techniques where two programmers work on a programming problem using only one computer [2]. Each programmer has a distinct role (i.e., driver and observer), thus the technique requires a dialog between the programmers in the pair. This dialog provides evidence of the programmer learning and can be unobtrusively recorded. The test-first technique forces the programmers to define the exact functionality of each method and the system will be automatically tested as it is developed. Refactoring transforms the source code to be more readable and more elegant [2].

Section 2 of the paper describes the self-directed learning model and other related work, while Section 3 explains the design of our replicated case study. The results of the replicated case study are discussed in Section 4 and the conclusions and the future work are presented in Section 5.

## 2. The Self-Directed Learning Model and Related Work

The self-directed learning theory was proposed by Rajlich and Xu [27] and it combines the constructivist learning theory [25] and Bloom's taxonomy of the cognitive domain [4].

Constructivist learning is based on the work of Piaget on child learning [25]. Piaget identified two main learning activities: *assimilation* and *accommodation*. Assimilation refers to the way learners deal with new knowledge and accommodation describes how learners reorganize their existing knowledge. Rajlich and Xu [26] divided assimilation into two separate activities: *absorption*, which occurs when learners add new facts to their knowledge and *denial*, which occurs when learners

reject them because the new facts do not fit in with the existing knowledge.

Accommodation was also divided into two separate activities. When the learners reorganize their knowledge to aid future absorption of new facts, we call their cognitive activity *reorganization*. We term it as *expulsion* when a part of the knowledge becomes obsolete or provably incorrect and the learners reject it.

Bloom and his colleagues identified six levels of learning known as the Bloom’s taxonomy of the cognitive domain [4]. The six levels include the simple recall or *recognition* of facts at its lowest level, then *comprehension*, *application*, *analysis*, *synthesis*, and *evaluation* [4]. For simplicity, we will refer in the paper to the Bloom’s taxonomy of the cognitive domain as Bloom’s taxonomy or as Bloom’s levels.

The self-directed learning model is a two dimensional model, composed on one dimension of the four cognitive activities and on the other dimension of the six levels of Bloom’s taxonomy (see Table 1).

**Table 1: The self-directed learning model. All four cognitive activities may take place at any of the Bloom’s levels during comprehension and programming**

	Assimilation		Accommodation	
	Absorption	Denial	Reorg.	Expulsion
Recognition	√	√	√	√
Comprehen.	√	√	√	√
Application	√	√	√	√
Analysis	√	√	√	√
Synthesis	√	√	√	√
Evaluation	√	√	√	√

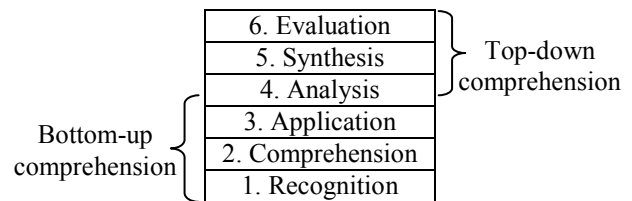
Several other cognitive models have been used for program comprehension. According to the top-down theory of comprehension [6], the programmer first defines a hypothesis that describes the program and then verifies this hypothesis. Further hypotheses may be required in order to build up a hierarchy of hypotheses to be verified. Letovsky [17] introduced the bottom-up theory, in which the programmer gathers together small chunks of source code to form higher levels of abstraction. These abstractions are recursively grouped together to produce a high level comprehension of a program. An as-needed approach was suggested by Littman et al. [18], where a programmer only looks at the code related to a particular problem or task. Similar to the self-directed learning model, all these models agree that existing knowledge is used in order to acquire new knowledge; this means that both assimilation and accommodation occur during comprehension. The top-down model mainly emphasizes on hypothesis and evaluation which are only two of the Bloom’s levels. Bottom-up model primarily strengthens (and is based on)

recognition, comprehension, and application - the three lower levels of Bloom’s taxonomy (see Figure 1).

Von Mayrhauser and Vans [30] observed that program comprehension is not a simple top-down or bottom-up process. They claimed that depending on tasks and familiarity of programmers with knowledge of domains and programming, programmers may apply various models [29, 31]. Programmers with more knowledge of domain prefer a more top-down approach than programmers with less domain knowledge. Developers with less programming knowledge use a more bottom-up approach in program comprehension. Based on their observations, Von Mayrhauser and Vans [29, 31] proposed an integrated program comprehension model which combines three sub-models: top-down, bottom-up, and as-needed. Based on this model, any of the three sub-models may be used at any time during the program comprehension process.

None of these models describe actual cognitive activities. Most models describe program comprehension as a simple procedure. According to Pennington [23] and Von Mayrhauser [29], programmers sometimes use the top-down, and sometimes use the bottom-up approach. None of the existing models can explain by itself the entire program comprehension process. Our self-learning theory is a more complete and detailed model.

Perkins and Martin [24] described the main difficulties faced by novices as “fragile knowledge” and “neglected strategies”. Novices obtain the knowledge, but do not know how to apply it. Gilmore [13] also studied the expert programming knowledge and recognized that expert programmers possess many strategies, while novices suffer not only from lack of knowledge, but also from lack of an adequate strategy to cope with the programming problems.



**Fig. 1. Comprehension strategies and their relationship with Bloom’s levels.**

Other differences between the novice and expert programmers were studied by Pennington [23], who found that experts use cross-referencing strategies whereas novices usually apply code-based and domain-based strategies. Davies [9] found that experts tend to rely much more upon the use of external sources than novices.

All this earlier research supports our view of these comprehension strategies, from the point of view of the

cognitive activities that take place during the process (see Figure 1).

### 3. Case Study Design

Martin and Koss developed a simple application for score keeping in bowling, using pair programming, test-first, and refactoring practices [19].

In this case study we replicated the pair-programming exercise published in [19] with two new pairs of programmers. We then analyzed the results together in order to provide answers for the questions raised in Section 1. This section presents the main elements of the replicated case study design.

#### 3.1. Dialog Based Protocol

Parnas [22] discussed the limitations of empirical studies in software engineering due to the Hawthorne effect [1]. The Hawthorne effect occurs when the subjects know that they are studied and that affects the outcome of the experiment. In order to reduce the Hawthorne effect, we employed a dialog based protocol in the design of our replicated case study.

Dialogue is a conversation between two people, which allows them to share ideas and learn from each other. The dialogue contains large amount of information and knowledge, which have been used as the raw data for studies in linguistic, psychology, sociology, literature, and other fields [5].

During pair programming, two programmers work side-by-side on design, algorithm selection, implementation, and testing, using a single computer. Since the programmers in the pair work together, they have to communicate what they think. The dialog allows the mental process to be easily verbalized and can reduce the Hawthorne effect, since programmers can easily forget that they are being studied when they talk freely.

If paired programmers are allowed to speak all that comes into their minds, one can record their communication and analyze it from the point of view of cognitive activities. We call this method *the dialog based protocol*. The dialog based protocol is similar to talk-aloud protocol, which refers to a method for psychological research that has been used to study cognitive activities in software engineering [28]. One major difference between the two protocols lies in the fact that in the talk-aloud protocol only one subject is required to speak out his/her own thoughts, whereas in the dialog based protocol, two people are verbally expressing their thoughts while completing the tasks. Both protocol techniques require the process to be recorded as audio or video for analysis.

#### 3.2. Recording Method

The dialogue of Martin and Koss was recorded and a reenactment of that pair programming episode was published in [19] and we conducted our preliminary study based on their dialogue [26].

In this case study, we videotaped the first replicating pair. For the second replicating pair, we used a free software “Microsoft producer” [20], which allowed us to capture computer screens and record voice at the same time. Neither videotaping nor the software voice recording disturbed the communication among programmer pairs; therefore the Hawthorne effect was minimized.

#### 3.3. Participants

Four graduate students from the Department of Computer Science at Wayne State University, who were taking a graduate software engineering course (i.e., CSC7110 - Software Engineering Environments), are classified as intermediate level programmers. They voluntarily joined the replicated case study as a part of their course projects. For participation in the experiment, they were rewarded with project credits. They had intermediate-to-large programming experience with C, C++, and Java, but they never used pair programming or test-first and refactoring practices.

#### 3.4. Material

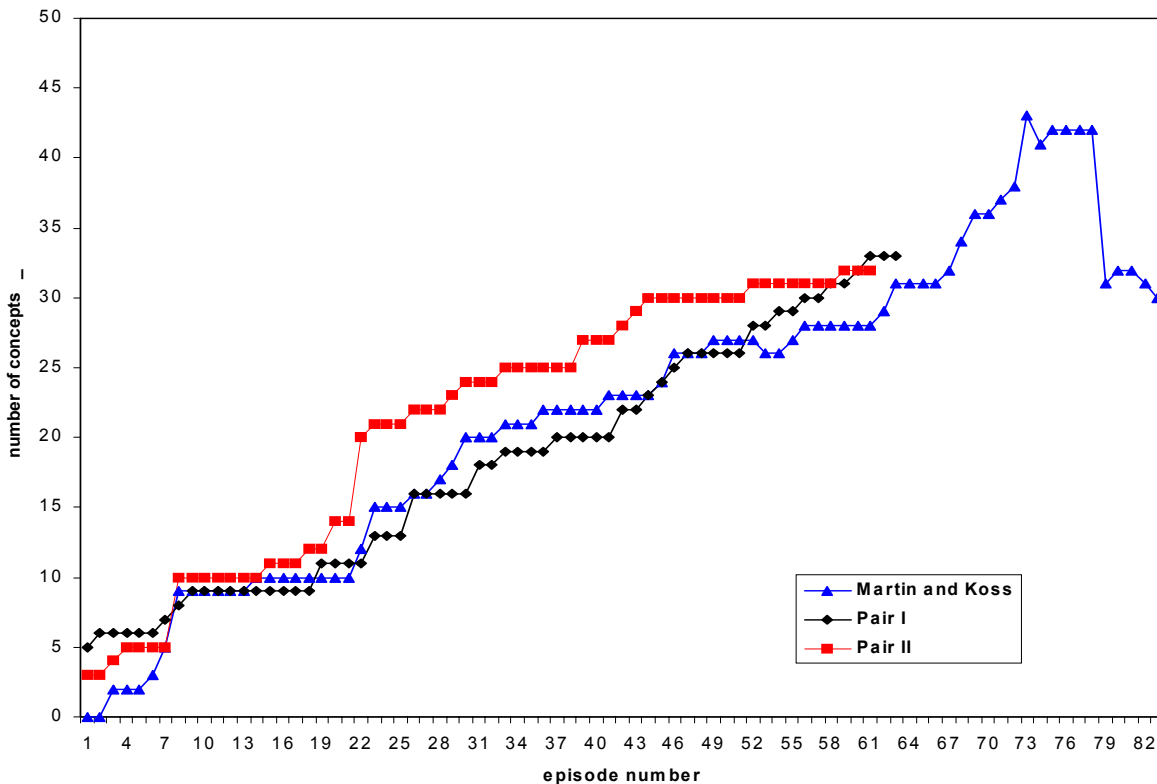
The task to be solved in the replicated case study is to implement an application which records the bowling scores for a bowling game. It requires the programmer pair to understand both domain concepts and programming concepts and we can make a comparison between the data collected during our replicated case study and the original work done by Martin and Koss [19]. Martin and Koss’ work was treated as experts’ work, since the pair of programmers in their study had over 15 years of programming experience and a considerable experience with pair programming, test-first, and refactoring practices.

Our intermediate programmers were not originally familiar with the bowling domain and did not know the solution of Martin and Koss.

#### 3.5. Procedures

Our replicated case study was carried over two semesters, in March 2004, with one pair and February 2005, with the other pair.

Since the programmers were new to pair programming, test-first programming and refactoring, and it is particularly difficult for novices to adopt test-first [21], we performed a short training session prior to the replicated case study. During the training session, the programmers were provided with reading materials on



**Fig. 2. The number of concepts in the knowledge of each pair during development.**

pair programming, test-first, and refactoring techniques, and they were asked to implement a simple program using the Eclipse environment and JUnit in order to understand the procedure and to be familiar with the tools.

One of the authors acted as the mentor, and recorded the replicated case study and the programmers were provided with a list of bowling scoring rules.

The replicated case study was divided into several sessions, each of which lasted about two hours.

### 3.6. Protocol Analysis

Once the recording sessions were finished, recordings were transcribed and then analyzed. The dialogue was decomposed into episodes using the following rule: a new episode begins when the programmers start discussion of a different concept. In some cases, long phrases contained information about several concepts; such phrases were kept in the same episode.

Each episode deals with specific concepts that are part of the knowledge. We simply classify the concepts as domain concepts and programming concepts in this study. Domain concepts belong to a specific domain which the program addresses [3], such as *strike* or *spare* in bowling,

while programming concepts belong to the knowledge of programming, such as the programming language, program development process, design decisions, and so forth. We count design decisions as programming concepts here, which result in the addition, modification, or deletion of classes, methods, or class variables.

Protocol analysis was employed for analyzing this observational data. The first analysis of the protocols involved assignment of activity types as specified by the self-directed learning model. We classified each episode according to the four cognitive activities (i.e., absorption, denial, reorganization, and expulsion) by using characteristic verbs defined in the self-directed learning model [27].

As a result of these activities, concepts were added or excluded from the existing knowledge. For example, during absorption, new concepts are added to the knowledge; during expulsion, concepts are excluded from the knowledge. The number of concepts in the knowledge was recorded for each episode (see Figure 2).

Then, the Bloom's levels (i.e., recognition, comprehension, application, analysis, synthesis, and evaluation) were assigned to each episode also through the use of characteristic verbs [8, 16]. The final data was then computed as the frequencies of occurrences in

episodes for each activity type and for each Bloom's level.

## 4. Results and Discussion

It took each pair around five hours to complete the task. The programs were complete and covered all significant test cases. Table 2 summarizes the characteristics of the developed programs by each pair and of their respective dialogues.

The distribution of cognitive activities and Bloom's levels are presented in Table 3, Table 4, and Table 5.

Figure 2 shows the changing numbers of concepts in the replicated case study and the Martin and Koss study. The final UML class diagrams [11] of the three programs are shown in Figure 4, Figure 5, and Figure 6 from the Appendix.

**Table 2: Main characteristics of the programs and dialogues developed by each pair.**

Items	Martin and Koss	Pair 1	Pair 2
Lines of code	215	246	222
Number of class members	36	26	25
Number of classes	3	2	3
Number of domain concepts discussed before coding started	9	2	3
Maximum number of concepts in the knowledge	43	33	32
Number of concepts in the knowledge at the end of the task	30	33	32
Number of refactorings	16	5	9
Number of episodes	84	64	62

### 4.1. Observations on Programming, Process, and Discourse

A comparison between the final programs constructed by Martin and Koss (expert programmers) and those by our programming pairs (intermediate programmers) shows that the former has a better design (see Figure 4, Figure 5, and Figure 6). Martin and Koss' class members are much more elegant and readable. They also separated a Scorer class from the class Game to calculate the scores for different cases, whereas our first programmer pair used the Bowling class to do everything. Pair 1 also used three arrays to store the scores, which turned out to be

```

"Ohh, we add test2Throw, but we have to
delete the old method for getScore ()
public int getScore () {
    Return 9;
}
since it does not do what we want"

```

Concepts: test2Throw, getScore  
 Activities: absorption and expulsion

**Fig. 3. An example of an episode from pair 1. This episode refers to multiple concepts and includes multiple cognitive activities. The episode was classified as *expulsion*.**

less efficient. The program implemented by our second pair is also hard to read with less meaningful variable names. However, both the expert and intermediate programmers created a similar number of test cases.

One programmer pair kept the two classes from the beginning to the end, while another pair defined two classes at the beginning and created one new class in a later stage. However, Martin and Koss [19] made a big change in terms of design decisions where they created five classes at the beginning but only kept two of them in the end. A big drop in term of numbers of concepts in the Martin and Koss' work is indicated in Figure 2. All these facts indicate that our intermediate programmers tend to keep the original design decisions due to lack of programming experience, while expert programmers more readily abandon obsolete or inadequate concepts.

The intermediate programmers were struggling in using test-first driven technique at the beginning. But they learned it quickly by looking at the supplied sample programs.

The intermediate programmers, in general, created the class design at the beginning of the process and kept it until the end. They tried to fit everything into their previous design decisions and were reluctant to change them. This fact is also expressed by the number of concepts they kept in the knowledge; this number did not decrease from one episode to another, indicating that little or no expulsion took place (see Table 4 and Table 5). Pair 2 did no expulsion at all, while pair 1 did expulsion in 4.69% (3) of the episodes. Yet the number of concepts in those episodes did not decrease.

The explanation behind this apparent contradiction lies in the structure of the episodes. The discourse in the Martin and Koss dialogue is more coherent and richer than that in the dialogue of our pairs. Our programmers often referred to more than one concept in one phrase; thus, such an episode will contain multiple activities where one could be absorption and the other could be expulsion (see Figure 3). Even though such an episode is categorized as expulsion, the number of concepts in the knowledge, at the end of the episode, may not change.

**Table 3: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for Martin and Koss.**

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Comprehension	20 (23.81%)	2 (2.38%)	5 (5.95%)	1 (1.19%)	28 (33.33%)
Application	9 (10.72%)	2 (2.38%)	5 (5.95%)	2 (2.38%)	18 (21.43%)
Analysis	17 (20.24%)	1 (1.19%)	5 (5.95%)	3 (3.57%)	26 (30.95%)
Synthesis	11(13.10%)	0 (0.00%)	1 (1.19%)	0 (0.00%)	12 (14.29%)
Evaluation	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	57 (67.87%)	5 (5.95%)	16 (19.04%)	6 (7.14%)	82 (100%)

The language used in the dialogue is also different. Martin and Koss' dialogue is edited from the original recorded data. Phrasing is more mature and better articulated. Our pairs used a lot of references to the source code (as in Figure 3), which are missing in the Martin and Koss dialogue.

Since our intermediate programmers were new to the refactoring technique, they applied it several times at the beginning of the process by only renaming the variables, but they did not extract methods or classes [12]. Experts often refactored the code when the application was close to completion and they performed more advanced refactorings such as reducing the scope of variables and extracting methods. This is consistent with the fact that Martin and Koss only added 40 lines of code at the end of process, but one of our programmer pairs added 141 lines of code and defined 11 new class members at the very end of the process. The fact is once again reflected by the evolution of the number of concepts in the knowledge (see Figure 2); Martin and Koss reduced the number of concepts towards the end of the process. Table 3, Table 4, and Table 5 also show that Martin and Koss' activities resulted in reorganization in more episodes (16 - 19.04%) than in the other pairs (5 - 7.82% and 9 - 14.52% respectively).

#### 4.2. Observations on Cognitive Activities

The experts discussed related domain concepts more extensively before they started coding, where they referred to 9 domain concepts. Intermediate programmers only talked about 2 and 3 domain concepts respectively, at the beginning. The lack of domain knowledge is often considered characteristic to novice programmers [13]. The fact that there are 84 episodes in the expert's work, and only 62 and 64 episodes respectively in the intermediate programmers' work, also indicates that the experts tend to switch concepts during the discussion more frequently than intermediate programmers.

In the replicated case study, the absorption activity dominates the programming process from the beginning to the end, at all levels of Bloom's taxonomy. There are more than 83% absorption activities in the work of our two pairs while there are only 67.87% in the Martin and Koss' work (see Table 3, Table 4, and Table 5), which could indicate that the intermediate programmers' lack knowledge compared to experts. A large amount of the absorption activity appeared in the last part in our two pairs' work, because our programmer pairs were getting familiar with the test-first driven technique and created more test cases at the end of the process.

Most absorption activities occur at the lower Bloom's levels such as comprehension and application, particularly for our intermediate programmers, who spent more than 43% and 56% respectively of total activities on absorption at comprehension and application levels.

**Table 4: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for Pair I.**

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Comprehension	18 (28.12%)	0 (0.00%)	2 (3.13%)	0 (0.00%)	20 (31.25%)
Application	18 (28.12%)	0 (0.00%)	3 (4.69%)	1 (1.56%)	22 (34.37%)
Analysis	8 (12.50%)	0 (0.00%)	0 (0.00%)	2 (3.13%)	10 (15.63%)
Synthesis	12 (18.75%)	0 (0.00%)	0 (0v%)	0 (0.00%)	12 (18.75%)
Evaluation	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	56 (87.49%)	0 (0.00%)	5 (7.82%)	3 (4.69%)	64(100%)

**Table 5: The distribution of the cognitive activities and Bloom's levels throughout the recorded episodes for Pair II.**

	Assimilation		Accommodation		Total
	Absorption	Denial	Reorganization	Expulsion	
Recognition	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Comprehension	12 (19.35%)	1 (1.61%)	0 (0.00%)	0 (0.00%)	13 (20.96%)
Application	15 (24.20.00%)	0 (0.00%)	7 (11.29%)	0 (0.00%)	22 (35.49%)
Analysis	13 (20.97%)	0 (0.00%)	2 (3.23%)	0 (0.00%)	15 (24.20%)
Synthesis	12 (19.35%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	12 (19.35%)
Evaluation	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	52 (83.87%)	1 (1.61%)	9 (14.52%)	0 (0.00%)	62 (100.00%)

In general, denial, expulsion, and reorganization activities occurred more frequently in the dialogue of Martin and Koss than in those of our two pairs. Unlike in the Martin and Koss' work, there is no concept which was intensively discussed as a potential programming concept but was ultimately rejected. It demonstrates the differences between intermediate and expert programmers where intermediate programmers do not have a deep discussion of the program design at the beginning, while at the same time experts try to create a clear picture of the major design decisions.

Four out of six Bloom's levels have been recognized in the replicated case study. More episodes were classified at the lower Bloom's levels (recognition, comprehension, and application) in the dialogues of our two pairs than that of Martin and Koss. This indicates that intermediate programmers spent more time in understanding and applying the knowledge while experts took part of the knowledge as granted and spent more time on analyzing the knowledge and generating test cases. This also coincides with that fact that our intermediate programmers had more absorption activities.

These results together with the comparison of the numbers of absorption activities, and the numbers of episodes classified at lower levels or higher levels of Bloom's taxonomy in our replicated case study, support Gilmore's observation [13] that novices not only lack knowledge, but also lack the strategies to apply the knowledge. Novices often take longer time to accumulate their knowledge than experts. Even if novices have as much knowledge as the experts, they may still need a longer time to finish their programming tasks, as they are not familiar with the strategies to use this knowledge. We do not know the time spent by Martin and Koss on implementing the bowling application; however, there are less lines of code in their program than in the programs written by our pairs.

One interesting fact we observed is that no episodes were classified at the top (i.e., evaluation) or at the bottom (i.e., recognition) of Bloom's taxonomy. Xu and Rajlich [32] previously identified all six levels of

Bloom's taxonomy in a case study on program debugging.

Recognition is the pre-requisite for the activities at comprehension and other higher levels. It is the main reason that it does not explicitly appear in our data. An additional reason is due to the way we split the dialogue into episodes (i.e., based on concepts rather than activities – see Section 3.6).

We believe that the programming activity in this case study was quite simple and the size of the domain and the resulting software were small. During more complex software engineering tasks (i.e., debugging, reverse engineering, reengineering, etc.) more hypothesis-driven activities are performed, which would be classified at the top level in Bloom's taxonomy (i.e., evaluation).

Although some differences exist between the results of the experts and intermediate programmers, there are a lot of similarities as well, such as the numbers of test cases created, the numbers of class members, the distributions of cognitive activities and Bloom's levels. We believe that these numbers reflect the nature of the completed application rather than the expertise and therefore they were the same for both experts and the intermediate programmers.

### 4.3. Limitations

Based on the observations we made during the replicated case study, we believe that the impact of the Hawthorne effect was minimal. Both intermediate programmer pairs almost or totally forgot that they were being taped. We are convinced that unlike think-aloud protocol which has significant Hawthorne effects since the programmers are explicitly forced to do something which may not happen in real world [14], pair programming provides a better way to study the cognitive activities during programming and software development.

While our case study can be easily replicated (thus independent confirmation of our results is possible), some of its features make the results hard to generalize:

- The problem solved by the programmers is relatively simple. In solving more complex problems, results may be different.
- The intermediate pairs did not have much experience working together, so their dialogues were rather short.
- The short dialogue and the simple problem make it hard to differentiate experts from novices.
- Subjectivity of the classification. Episodes were classified manually by one of the authors, as explained before. The classifications were discussed and consensus was not always reached among all the authors.
- Splitting the episodes based on concepts. A different splitting criterion (let's say based on activities rather than concepts) might yield different results.

## 5. Conclusions and Future Work

Our replicated case study showed that the self-directed learning model can be used to study the cognitive activities during incremental software development. We also found that the Martin and Koss exercise is duplicable with different types of programmers.

During incremental software development with the test-first approach, absorption is the dominant activity behind the process. Reorganization often occurs, but denial and expulsion occasionally appear. Four out of six of Bloom's levels were identified in the replicated case study.

In terms of differences between experts and intermediates, experts tend to discuss broadly on the problems and related domain concepts, while intermediates try to deal with individual concepts before discussing new ones. Experts are also willing to recognize and reconsider inappropriate design decisions, but intermediates seem to keep using them. Experts also apply well test-first and refactoring techniques during programming. The program written by experts has a better design and it is more efficient. Experts seem to, more or less, take knowledge as granted and spend more time to synthesize the knowledge and generate hypothesis.

In the future, we plan to extend the case study by involving additional programming pairs and study the cognitive variations among them. We also plan to conduct research on incremental change in large programs and to evaluate the impact of the size of programs on the cognitive activities of the programmers.

Other future work includes creation of a knowledge base that will capture the changing state of the programming knowledge during the program development and hence help new programmers to join in the middle of the project. We are also investigating

automatic classification of dialogue episodes using information retrieval methods.

## References

- [1] Adair, G., "The Hawthorne effect: a reconsideration of the methodological artifact", *Journal of Applied Psychology*, vol. 69, no. 2, 1984, pp. 334-345.
- [2] Beck, K., *Extreme Programming Explained*, Massachusetts, Addison-Wesley, 2000.
- [3] Biggerstaff, T. J., Mitbender, B. G., and Webster, D. E., "Program understanding and the concept assignment problem", *CACM*, vol. 37, no. 5, May 1994, pp. 72-82.
- [4] Bloom, B. S., "Taxonomy of Educational Objectives: The Classification of Educational Goals: Handbook, I, Cognitive Domain": New York, Toronto, Longmans, Green, 1956.
- [5] Bohm, D. and Nichol, L., *On Dialogue*, Brunner-Routledge, 1996.
- [6] Brooks, R., "Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies*, vol. 18, no. 6, 1983, pp. 543-554.
- [7] Curtis, B., "A field study of the software design process on large systems", *Communication of the ACM*, vol. 13, no. 11, 1988, pp. 1268-1287.
- [8] Dalton, J. and Smitd, D., *Extending children's special abilities: strategies for primary classrooms*, Melbourne, Victorian Ministry of Education, 1986.
- [9] Davies, S. P., "Models and theories of programming strategy", *International Journal of Man-Machine Studies*, vol. 39, no. 2, 1993, pp. 237-267.
- [10] Fischer, G., McCall, R., Ostwald, J., Reeves, B., and Shipman, F., "Seeding, evolutionary growth and reseeded: supporting the incremental development of design environments", in Proceedings of Conference on Computer-Human Interaction (CHI'94), Boston, MA, 1994, pp. 292-298.
- [11] Fowler, M., *UML Distilled*, Addison Wesley Longman, 1997.
- [12] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [13] Gilmore, D. J., "Expert programming knowledge: a strategic approach", *Psychology of Programming*, 1990, pp. 223-233.
- [14] Gilmore, D. J., "Methodological issues in the study of programming", *Psychology of Programming*, 1990, pp. 83-97.
- [15] Henninger, S., "Tools supporting the creation and evolution of software development knowledge", in Proceedings of International Conference on Automated Software Engineering (ASE'97), 1997, pp. 46-53.
- [16] Huitt, W., "Bloom et al.'s taxonomy of the cognitive domain: educational psychology interactive", Valdosta, GA: Valdosta State University, <http://chiron.valdosta.edu/whuitt/col/cogsys/bloom.html>, 2004.

[17] Letovsky, S. and Soloway, E., "Delocalized plans and program comprehension", *IEEE Software*, vol. 19, no. 3, May 1986, pp. 41 - 48.

[18] Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E., "Mental models and software maintenance", in *Empirical Studies of Programmers*, Soloway, E. and Iyengar, S., Eds., Norwood N.J. Albex Publisher Coop., 1986, pp. 80 - 98.

[19] Martin, R. C., *Agile Software Development, Principles, Patterns, and Practices*, Massachusetts, Addison Wesley, 2002.

[20] Microsoft, "Microsoft Producer for Microsoft Office Powerpoint 2003", <http://www.microsoft.com/windows/windowsmedia/technologies/producer.aspx>.

[21] Muller, M. and Tichy, W., "Case study: extreme programming in a university environment", in Proceedings of 23rd International Conference on Software Engineering, Toronto, Canada, May 2001, pp. 537-544.

[22] Parnas, D. L., "The limits of empirical studies of software engineering", in Proceedings of 2003 International Symposium on Empirical Software Engineering, 2003, pp. 2-5.

[23] Pennington, N., "Comprehension strategies in programming", in *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, and Soloway, E., Eds., Norwood, NJ: Ablex. Publisher Coop., 1987, pp. 100-113.

[24] Perkins, D. N. and Martin, F., "Fragile knowledge and neglected strategies in novice programmers", in *Empirical Studies of Programmers*, Soloway, E. and Iyengar, S., Eds., Norwood N.J. Albex Publisher Coop., 1986, pp. 213-229.

[25] Piaget, J., *The Construction of Reality in the Child*, New York, Basic Books, 1954.

[26] Rajlich, V. and Xu, S., "Analogy of incremental program development and constructivist learning", in Proceedings of the Second IEEE International Conference on Cognitive Informatics, London, UK, 2003, pp. 142-150.

[27] Rajlich, V. and Xu, S., "Constructivist learning during software development", *IEEE Transactions on Systems, Man and Cybernetics* 2005, to appear.

[28] Von Mayrhauser, A. and Lang, S., "A coding scheme to support analysis of software comprehension", *IEEE transactions on software engineering*, vol. 25, no. 4, 1999, pp. 526-540.

[29] Von Mayrhauser, A. and Vans, A., "Program understanding behavior during adaptation of large scale software", in Proceedings of the Sixth International Workshop on Program Comprehension, Los Alamitos, CA, 1998, pp. 164-172.

[30] Von Mayrhauser, A. and Vans, A. M., "From program comprehension to tool requirements for an industrial environment", in Proceedings of Second Workshop on Program Comprehension, Capri, Italy, 1993, pp. 78-86.

[31] von Mayrhauser, A. and Vans, A. M., "Program understanding during software adaptation tasks", in Proceedings of International conference on software maintenance, 1998, pp. 316-325.

[32] Xu, S. and Rajlich, V., "Cognitive process during program debugging", in Proceedings of Third IEEE International Conference on Cognitive Informatics, Victoria, BC, August 16-17 2004, pp. 176-182.

## Appendix: UML Diagrams for the Implemented Programs

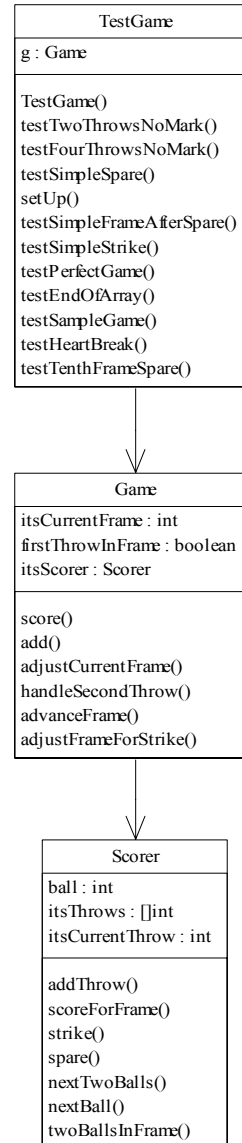


Fig. 4. UML class diagram of Martin and Koss' final program.

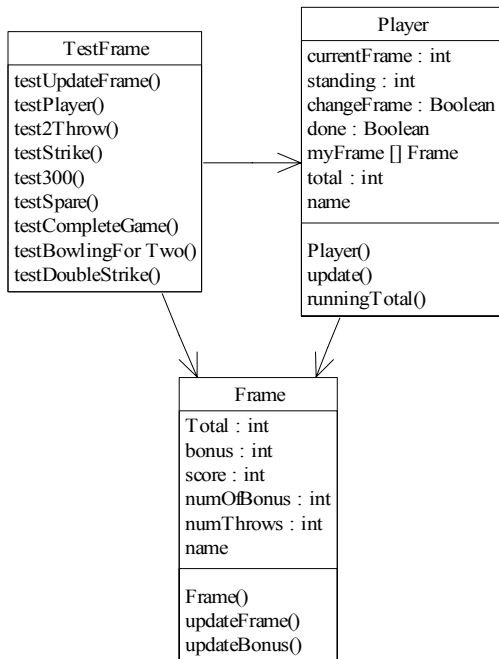


Fig. 5. UML class diagram of Pair 2's final program.

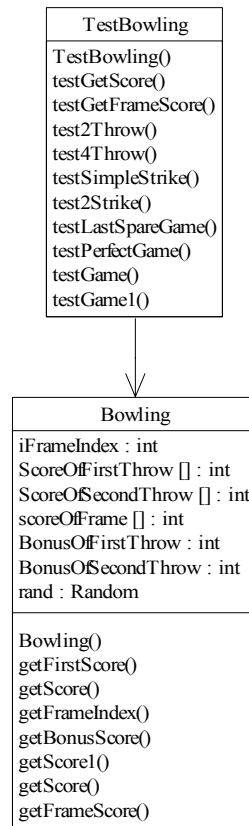


Fig. 6. UML class diagram of Pair 1's final program.