Kong, S.C., Ogata, H., Arnseth, H.C., Chan, C.K.K., Hirashima, T., Klett, F., Lee, J.H.M., Liu, C.C., Looi, C.K., Milrad, M., Mitrovic, A., Nakabayashi, K., Wong, S.L., Yang, S.J.H. (eds.) (2009). Proceedings of the 17th International Conference on Computers in Education [CDROM]. Hong Kong: Asia-Pacific Society for Computers in Education.

J-LATTE: a Constraint-based Tutor for Java

Jay HOLLAND, Antonija MITROVIC, Brent MARTIN
Intelligent Computer Tutoring Group, University of Canterbury, New Zealand
jah130@student.canterbury.ac.nz

Abstract: We present J-LATTE, a constraint-based intelligent tutoring system that teaches a subset of the Java programming language. J-LATTE supports two modes: *concept* mode, in which the student designs the program without having to specify contents of statements, and *coding* mode, in which the student completes the code. We present the style of interaction with J-LATTE, its interface, domain model and the student modeling approach. We also report the results of a study we conducted in an introductory programming course. Although we did not have enough participants to obtain statistical significance, the results show very promising trends indicating that students learned the constraints.

Introduction

Many students find it difficult to grasp the core programming concepts, which are commonly taught in the context of a specific programming language, such as Java. Intelligent Tutoring Systems (ITSs) have been shown to support learning effectively in many areas, such as mathematics [3], physics [9], SQL queries and database design [5]. J-LATTE (Java Language Acquisition Tile Tutoring Environment) is our attempt to teach Java to students. The system represents the domain knowledge in terms of constraints [6]. Constraints do not restrict the student to a set path while forming a solution: provided the constraint set sufficiently covers the domain, any valid solution will be recognized as such by the system. The only exception to this is that in the interest of teaching good practice J-LATTE contains constraints that enforce good programming style. We begin by discussing how students solve problems in J-LATTE. Section 2 presents the student modeling approach used. A preliminary study is reported in Section 3. We conclude by comparing J-LATTE to other programming ITSs and discuss future work.

1. Learning to Program with J-LATTE

Students need to learn how to design programs as well as learn the syntax of the language. Often this is done all at once, which is arguably beneficial: design theory without the practical side leaves out the relevant context that gives a student greater understanding, whereas a practical curriculum without the design side enables the student to know the tools, but not how to combine them in a meaningful way. J-LATTE tutors students in the skills needed to manage program complexity in basic programming tasks (defined here as tasks that do not require architectural design). The level of complexity we are interested in is when students must deal with writing syntactically correct code, whilst also having to think about higher-level programming concepts. Our approach is to lessen the cognitive load and also teach simple skill separation by encouraging (but not forcing) the student to focus on one of these tasks at a time. We do so by dividing the solution task explicitly into two modes. In J-LATTE, students design programs initially with certain abstract concepts from

Java, without worrying about the code level complexity, in what we call concept mode. At this level the student develops the overall solution outline, but does not enter any actual code. Of course, working at the code level is still very important for learning programming, and in J-LATTE the student is still required to enter code at some point, by entering coding mode. This occurs after the student has created a complete or partial solution outline using the abstracted concepts; at the very least the student must have one concept in their solution outline before proceeding to coding mode. The student can then work at the code level whenever they feel comfortable, and can move between the two modes freely.

The level we chose for the concept abstraction was at the *statement* and *block* level, because they provide naturally occurring divisions in Java. Each concept either represents a whole Java statement (e.g. *Assignment*, *Declaration*), or a Java block, which may contain other statements (*For Loop*, *If Statement*). J-LATTE covers the first half of a semester-long introductory Java course, and includes the following concepts: *Declaration*, *Assignment*, *Print Statement*, *Return Statement*, *If Statement* and *For Loop*. These concepts are represented in the student interface in the form of tiles located at the bottom of the page, which the student can position at any location within the problem-solving area (see Figure 1). Block tiles are distinguished from statement tiles by a larger size and a darker border.

The student starts by selecting tiles and dragging them to the solution area. To enter the code, the student clicks the Refresh Code Boxes button, which displays text boxes in which they enter the actual code. For example, the solution in Figure 1 consists of a *For Loop* tile, which contains an assignment tile into which the student has entered "x=7".

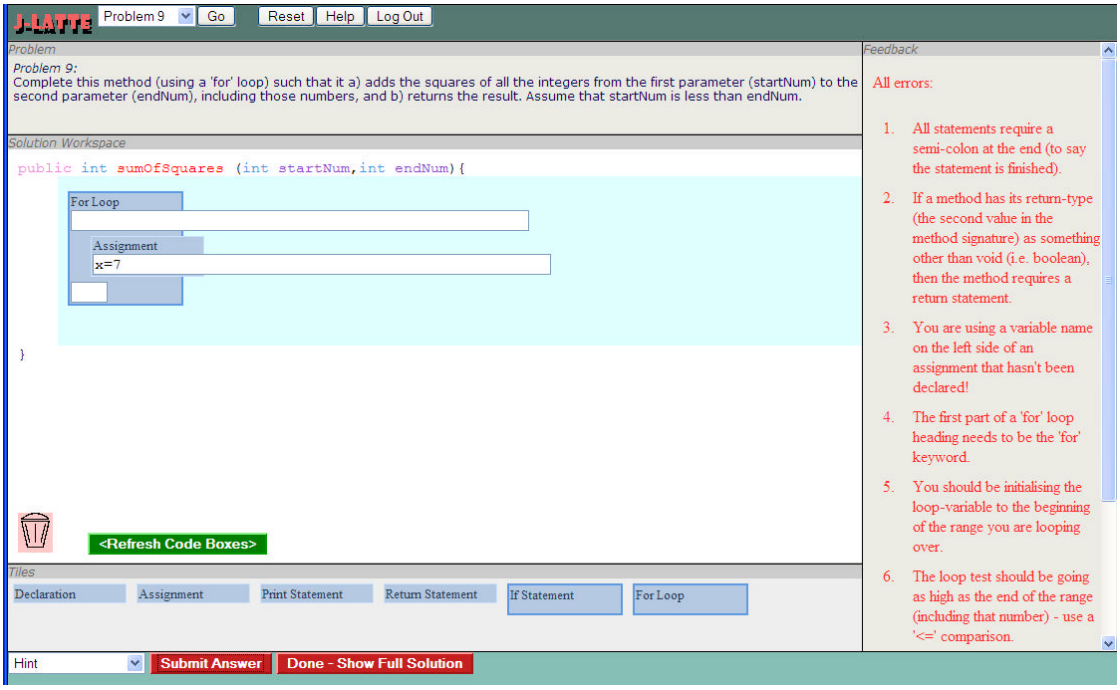

Figure 1. The interface of J-LATTE

The system includes three types of problems: *expression printing*, *predicate method* and *simple iteration*. A problem of the first type requires a student to implement a method that prints out a value, such as "*Complete this method such that, when run, it will display the arguments multiplied together*." A predicate method problem requires a method that returns a Boolean value based on a set of conditions (e.g."*Complete this method such that it returns true if the length of the given name is less than 20*.") A simple iteration problem requires a method that returns a value that must be calculated using a loop or nested loops, such as "*Implement the method sumOfPowers that returns the sum of each of the values from argument 'lowest' to argument 'highest' (including those values) raised to the argument 'power'*." Each problem is presented with the context, which is a code fragment provided to

the student. For example, the context might be the beginning and end of a *for* loop, or a method outline (signature and braces). The screenshot Figure 1 involves a problem of the third type (simple iteration) and the problem context consists of the method signature. At any time the student can ask for feedback by submitting the solution. J-LATTE thus provides feedback on demand, leaving the student in control of the instructional session. The feedback is provided in the feedback pane on the right of the page, and is red if the student has made errors.

2. Evaluating Students' Solutions

J-LATTE represents the domain knowledge using constraints, which describe features of correct solutions [6]. A constraint consists of a relevance condition that identifies problem states for which the constraint is important, and a satisfaction condition that specifies the features the solution must have to be correct. Constraints in J-LATTE are categorized as *syntax*, *semantic* and *style* constraints. Syntax constraints check that the student's solution consists of valid Java code, such as "*Each assignment must contain a valid expression on the right-hand side.*" Semantic constraints check whether the student's solution is the correct answer for the given problem. For example, an example semantic constraint is "*If the problem requires a function to be applied to a range of values, the solution must contain a loop.*" Each constraint evaluates one small fragment of the solution; in the case of the previous example, there are other constraints that check that the loop is in the correct place, the counter is initialized and incremented correctly, and the loop condition is correct. Style constraints encourage the student to follow good practice, such as "*The return statement should be at the end of a method.*" Semantic constraints compare the student's solution to the formal specification of the problem, which specifies a list of *requirements*. The specification of the problem shown in Figure 1 is:

```
(sum-of-function-over-a-range :range (:from (method-arg :name "startNum")
                                     :to (method-arg :name "endNum")) :function square)
```

This problem requires the *sum-of-function-over-a-range* pattern, and then specifies the lower and upper boundary for the range. Finally, the function used when summing is specified (*square*). Therefore, the student may use various types of loops, as well as various ways of specifying the square function and the loop condition. When the student submits his/her solution, the student modeller evaluates it and produces the lists of relevant, satisfied and (possibly) violated constraints. There are several levels of feedback in J-LATTE. *Simple Feedback* only indicates whether the solution is correct or not, *Hint* shows the messages corresponding to the first violated constraint, while *All Errors* displays a list of all the errors (shown in Figure 1). If a student's submission is correct, the student will be congratulated, and asked to choose another problem.

3. Preliminary Evaluation Study

A study was carried out with 26 volunteers from an introductory programming course at the University of Canterbury in 2008. The students used J-LATTE during scheduled labs for the course. The labs were 110 minutes long, during which time the students sat a pre-test, interacted with the system, and then sat a post-test and filled in a user questionnaire. The maximum interaction time was therefore 90 minutes. The study was performed in the sixth week of the course, immediately after all the concepts necessary for the problems J-LATTE contains were covered in lectures. The experimental group used the full system, while the

control group used a cut-down version that simulates the classroom condition by not providing feedback on errors. The control group could therefore only request the full solution in order to compare it to their own. In both groups students could not continue working on the problem after seeing the full solution. J-LATTE contained 44 syntax, 43 semantic and two style constraints, and 11 problems (3 display, 4 predicate and 4 loop problems). Table 1 reports means (standard deviations in brackets) from the study. Data about two participants were excluded because they spent less than 10 minutes with the system. The pre/post tests had five questions each, with a maximum mark of 5. There was no significant difference on the pre-test performance of the two groups.

Table 1. Some statistics about the study. The post-test score for the control group includes six students only.

	Control	Experimental	Significant
Participants	10	14	
Interaction time (min)	45 (18)	62 (18)	$p < 0.02$
Attempted problems	9.5 (2.2)	7.8 (2.3)	$p < 0.04$
Solved problems	2 (0.8)	5.6 (2.6)	$p < 0.01$
Pre-test scores	2.6 (0.9)	2.5 (1)	not significant
Post-test scores	3.8 (0.9)	2.8 (1.2)	N/A

The control group students spent less time with J-LATTE and attempted more problems than their peers from the experimental group (both differences are significant). Experimental group students, on the other hand, solved significantly more problems than the control group, probably because of the help they received via feedback on errors. Four control group participants did not submit post-tests, resulting in only 6 submissions from this group. As this number is small, and also less than a half of the experimental group, we did not apply statistical tests to the post-test results. Both groups improved from pre- to post-test, but the improvement was not significant.

Table 2. Mean scores from the questionnaire (scale from 1 to 5)

Question	Control	Experimental
Overall quality (poor to excellent)	3.5 (0.6)	3.5 (0.9)
Impression (terrible to wonderful)	3.2 (0.5)	3.3 (0.9)
Impression (difficult to easy)	3.2 (1.1)	2.9 (0.9)
Impression (boring to fun)	3.6 (0.7)	3.4 (0.9)
Feedback quality (poor to excellent)	2.3 (0.6)	3.7 (1.1)

The user questionnaire replies are summarized in Table 2. Most of the ratings are similar, except for feedback; the experimental group students clearly appreciated getting help on their solutions. We analyzed how students learned constraints. Figure 2 shows the learning curve for the experimental group students, as a plot of the probability of error on each occasion of use, averaged over all constraints and all students. The data points show a regular decrease of constraint violations over time, thus proving that students are learning constraints. The probability of constraint violation drops by 40% from the initial value of 0.27 by the 15th

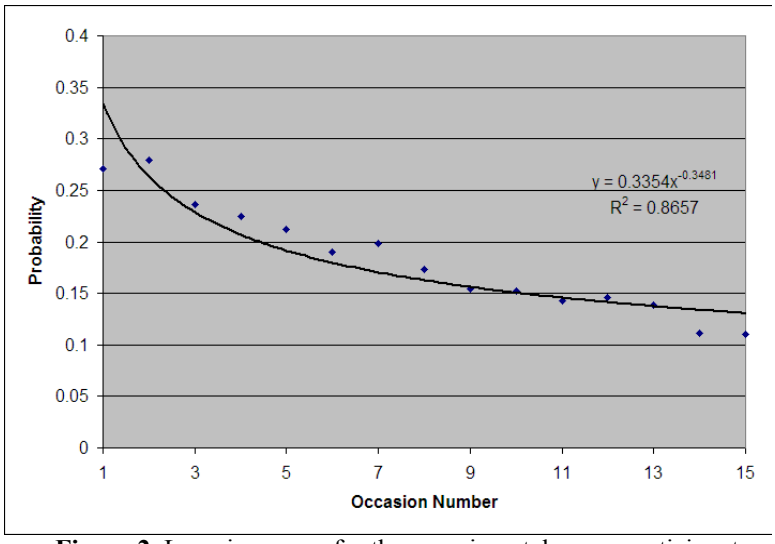

Figure 2. Learning curve for the experimental group participants

attempt. The correlation between the number of constraints learned during the session and the number of feedback messages received is 0.78, indicating that students do learn from feedback on errors.

4. Related Work and Conclusions

Programming is a challenging task for ITSs. Systems such as PROUST [2], Lisp Tutor [7], JITS [8] and RoboProf [1] give students experience with writing complete programs, whereas others, like [4], focus on tutoring a particular skill only. Solutions to programming problems, except in trivial cases, are not singular; values can be changed, different operators can be used, and even structural changes can be made, and the solution still satisfies the problem requirements. Accepting all correct solutions (and rejecting all incorrect solutions) is a difficult task, and not one that offers a single obvious answer. Lisp Tutor solves this problem by not allowing true free-form coding; the system forces the student to complete the code top-down, from left to right, and provides immediate feedback on errors. This reduces the complexity of evaluation, but also reduces the student's freedom. J-LATTE does not impose any restrictions on how the code is written: the student is free to work on any part of the program they like. RoboProf does not give the fine-grained feedback that is important for supporting learning; it also analyses the output of the program only and not the program itself, and may accept incorrect solutions. PROUST decomposes the goals of the problem into programming plans, and then matches these plans to the input. This idea is intuitive; what determines the assessment accuracy is the performance of matching the plans to parts of the solution, and correctness of the plans themselves. JITS employs a similar method; it creates a functional decomposition. J-LATTE enables novice programmers to practise their skills and get feedback on their solutions. The original contribution of J-LATTE is the distinction between the two modes ("concept" and "coding"). This feature makes the evaluation process much simpler than the one used in JITS and PROUST.

The preliminary evaluation of J-LATTE is promising: although we did not have enough participants, there is evidence that students do learn constraints and appreciate the feedback from the system. We plan to conduct a bigger study in 2009, as well as to investigate whether the approach taken will scale to more complex problems.

References

[1] Daly, C., & Horgan, J. (2004). An automated learning system for Java programming. *IEEE Transactions on Education*, *47*(1), 10–17.

[2] Johnson, W. L., & Soloway, E. (1984). PROUST: Knowledge-based program understanding. *Proc. 7th Int. Conf. Software Engineering*, 369–380.

[3] Koedinger, K. R., Anderson, J. R., Hadley, W. H., & Mark, M. A. (1997). Intelligent tutoring goes to school in the big city, *Artificial Intelligence in Education, 8*, 30–43.

[4] Kostadinov, R., & Kumar, A. (2003). A tutor for learning encapsulation in C++ classes. *Proc. ED-MEDIA*, 1311–1314.

[5] Mitrovic, A., Martin, B., & Suraweera, P. (2007). Intelligent tutors for all: Constraint-based modeling methodology, systems and authoring. *IEEE Intelligent Systems*, *22*(4), 38–45.

[6] Ohlsson, S. (1994). Constraint-based student modelling. *In Proc. of Student Modelling: the Key to Individualized Knowledge-based Instruction*, 167–189.

[7] Reiser, B. J., Anderson, J. R., & Farrell, R. G. (1985). Dynamic student modeling in an intelligent tutor for Lisp programming. *Proc. IJCAI*, 8–13.

[8] Sykes, E. R., & Franek, F. (2003). An intelligent tutoring system prototype for learning to program Java. *Proc. 3rd IEEE Int. Conf. Advanced Learning Technologies*, 485–486.

[9] VanLehn, K. et al. (2005). The Andes physics tutoring system: Lessons learned. *Artificial Intelligence in Education, 15*(3), 147–204.