Stichting NIOC en de NIOC kennisbank

Stichting NIOC (www.nioc.nl) stelt zich conform zijn statuten tot doel: het realiseren van congressen over informatica onderwijs en voorts al hetgeen met een en ander rechtstreeks of zijdelings verband houdt of daartoe bevorderlijk kan zijn, alles in de ruimste zin des woords.

De stichting NIOC neemt de archivering van de resultaten van de congressen voor zijn rekening. De website www.nioc.nl ontsluit onder "Eerdere congressen" de gearchiveerde websites van eerdere congressen. De vele afzonderlijke congresbijdragen zijn opgenomen in een kennisbank die via dezelfde website onder "NIOC kennisbank" ontsloten wordt.

Op dit moment bevat de NIOC kennisbank alle bijdragen, incl. die van het laatste congres (NIOC2025, gehouden op donderdag 27 maart 2025 jl. en georganiseerd door Hogeschool Windesheim). Bij elkaar zo'n 1500 bijdragen!

We roepen je op, na het lezen van het document dat door jou is gedownload, de auteur(s) feedback te geven. Dit kan door je te registreren als gebruiker van de NIOC kennisbank. Na registratie krijg je bericht hoe in te loggen op de NIOC kennisbank.

Het eerstvolgende NIOC vindt plaats op 18 maart 2027 in Arnhem en wordt georganiseerd door HAN University of Applied Sciences.

Reacties over de NIOC kennisbank en de inhoud daarvan kun je richten aan de beheerder:

R. Smedinga kennisbank@nioc.nl.

Vermeld bij reacties jouw naam en telefoonnummer voor nader contact.

ARTIKEL

PHP Tutor System

Recommendations for the Functionality of an Intelligent Tutor System (ITS) for the Programming Language PHP

Door: Gerrie van Leeuwen, voortgezet onderwijs.
Trefwoorden: leren programmeren, Intelligent Tutor System, PHP.

MSc Thesis: Gerrie van Leeuwen (Edited Partial Reprint); Master of science education and communication of computer science / informatics; Utrecht University; the Netherlands.

Learning to program is difficult for novice programmers. Human tutoring often helps, but is time consuming and not always available. An Intelligent Tutoring System (ITS) may be of help here. An ITS involves selecting a problem-solving interface, designing a cognitive model for solving problems in that environment and building instruction around the productions in that model. The current research gives recommendations for the cognitive model and the instruction of an ITS for the programming language PHP. For that purpose, we need to find out how novice programmers solve programming problems in PHP. Novice programmers are studied by letting them think aloud and logging their voice and screen output while they are constructing PHP programs. We look at the lines of code they develop, and the problems they encounter. In particular, we determine which problems are specific for PHP. Then, we look at their questions and the examples they use. From this we derive production rules, 'Frequently Asked Questions" which can be answered by the tutor, and examples which can support the students when solving specific programming problems. The recommendations for an ITS for the programming language PHP consist of these production rules to inform the cognitive model, the 'Frequently Asked Questions" together with examples to support the instruction. Finally we show an interactive session with a hypothetical ITS for PHP in which our recommendations are implemented.

1. Introduction

Computer programming at an elementary level is part of the high school curriculum. There is quite some evidence for low learning outcomes after relatively short elementary programming courses of 10-50 lessons (Linn, 1985; Pea & Kurland, 1984). After these courses, most students still have an incomplete or incorrect mental model of the working of a computer (DuBoulay, 1986; Pea, 1986), a fragile knowledge base related to the basic commands and syntax of the programming language (Perkins & Martin, 1986; Putnam, Sleeman, Baxter & Kuspa, 1986; Sleeman, Putnam, Baxter & Kuspa, 1986), a serious lack of programming language templates or programming plans (Dalbey & Linn, 1985), and ill-developed procedural skills, such as for planning the solution or testing and debugging the program (Kurland, Pea, Clement & Mawby, 1986; Van Merriënboer & Paas, 1990).
In most cases, human tutoring is used to help students to learn programming. Human tutoring has a long history in education and has been shown to be very effective. Bloom's studies of human tutors (Bloom, 1984) showed that expert human tutoring resulted in excellent learning gains. He showed that tutored students performed two standard deviations better than the students in the control

group. Furthermore, he noticed reduced learning differences: 90% of the tutored students performed as well as the top 20% of the students in the control group. Additionally, human tutoring improves students' attitude towards learning, interest and motivation (Anania, 1983). Unfortunately, human tutoring is expensive and cannot scale effectively to a large number of students. Intelligent Tutoring Systems (ITS) may be able to solve these disadvantages. ITSs are a product of educational theory, Artificial Intelligence (AI), and computer-human factors and attempt to provide the benefits of human tutoring to an unlimited number of students and can be used inside and outside the classroom (Phillips, 2011).

Carnegie Mellon University (CMU) has developed intelligent programming tutors for the programming languages Lisp, Prolog and Pascal and Cognitive Tutors for Mathematics. Students working with cognitive tutors completed problem solving activities in as little as 1/3 of the time needed by students working in conventional problem solving environments and performed as much as a letter grade better on post-tests than students who completed standard problem solving activities (Anderson et al., 1995).These tutors are based on the ACT theory. The tutors for Prolog and Pascal are not available and cannot be studied for this research.

At many secondary schools in the Netherlands, the programming language PHP is used to teach students programming. First, students learn to use HTML to develop web pages. After that, they learn the programming language PHP to develop dynamic web pages. Next, PHP is used to connect with a database system via SQL. HTML, CSS, PHP and SQL form a natural sequence of subjects and fit the curriculum. PHP is open source software and commonly used. At this moment there is no intelligent tutor system for PHP. Given the possible positive impact of an ITS, it would be useful to develop an ITS for PHP. Many students worldwide might profit from such a system. The goal of this research is to provide recommendations for the functionality of an ITS for the programming language PHP.

2. Theoretical background

In this chapter we look at the means-ends problem solving theory of Newell and Simon (1972) and the ACT learning theory of Anderson (1983, 1993). Then we give an overview of the aspects which are relevant for learning to program and the development from novice programmer to expert programmer (van Merriënboer & Paas, 1990). Next, the known problems (Spohrer& Soloway, 1989) and the capabilities and behaviour of novice programmers will be discussed (Robins, Rountree & Rountree, 2003). Finally, we look at the general aspects of the working of intelligent tutor systems (vanLehn, 2006) and give eight general principles for the design of an ITS (Anderson et al., 1987, 1989, 1990).

2.1. Basic concepts in problem solving and learning

The CMU Intelligent Tutoring Systems are based on the ACT (Adaptive Character of Thought) theory of cognition (Anderson, 1983, 1993). The ACT theory is a cognitive architecture: a theory about how human cognition works. From a distance, ACT looks like a programming language; however, its constructs reflect assumptions about human cognition. These assumptions are based on numerous facts derived from experiments in cognitive psychology. The ACT theory makes a distinction between declarative knowledge, which encodes factual knowledge, and procedural knowledge, which encodes many of the cognitive skills including problem solving skill. The theory assumes that problem solving takes place basically within a means-ends problem solving structure (Newell & Simon, 1972).

The concept of a problem-solving state is probably the most basic term in the Newell and Simon characterization of problem solving. A problem solution can be characterized as the solver beginning in some initial state of the problem, traversing through some intermediate states, and arriving at a state that satisfies the goal. If the problem is finding one's way through a maze, the states might be the various locations in the maze. The second key construct is that of a problem-solving operator. An operator is an action that transforms one state into another state. In the maze the obvious operators are going from one location to another which may bring the solver closer to the goal state. Together the concepts of state and operator define the concept of a problem space. At any state some operators apply, each of which will produce a new state, from which various operators can apply producing new states, and so forth. Within the problem-space conception, the problem in problem solving is search, which is to find some sequence of problem solving operators that will allow traversal in the problem space between the current state and a goal state. A flowchart of the Means-Ends Analysis is given in figure 1. Flowchart I breaks a problem down into a set of differences and tries to eliminate each. Flowchart II searches for an operator relevant eliminating a difference.

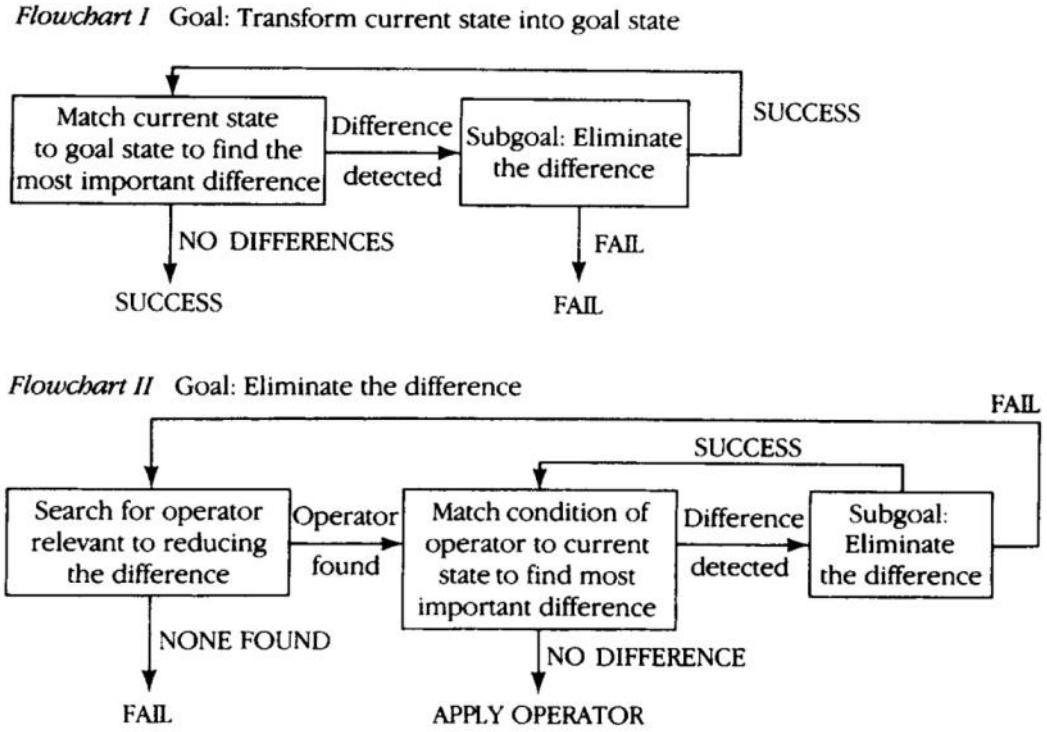

Figure 1. Application of the Means-Ends Analysis.

ACT is a theory of the origin and nature of the problem-solving operators that feed the means ends analysis. It assumes that when a problem solver reaches a state for which there is no adequate problem solving operators, the problem solver will search for an example of a similar problem solving state and try to solve the problem by analogy to that example. There is substantial evidence that a subject's early problem solving is strongly influenced by analogy to similar examples (e.g., Pirolli, 1985; Ross, 1984). Anderson and Thompson (1989) have developed a simulation model of this analogy process.

This initial stage of problem solving is called the interpretative stage. It often requires recalling specific problem-solving examples and interpreting them. The memories retrieved are declarative

memories. However, it is not necessary that the long-term memory is involved. For instance, students use examples in a mathematics section to solve a problem given at the end of the section without ever committing the examples to memory.

The interpretive stage can involve substantial verbalization as the learner rehearses the critical aspects of the example from which the analogy derives. There is a dropout of verbalization that is associated with the transition from this interpretive stage to a stage where the skill is encoded procedurally. Knowledge compilation is the term given to the process of transiting from the interpretive stage to the procedural stage. Procedural knowledge is encoded in terms of production rules that are condition-action pairs. For example, the following sentence and example of equation solving in an algebra text would be encoded declaratively:

> When the quantities on both sides of an equation are divided by the same value, the resulting quantities are equal. For example:
>
> IF we assume 2X = 12, THEN we can divide both sides of the equation by 2, and the two resulting expressions will be equal, X = 6.
>
> IF the goal is to solve an equation for variable X and the equation is of the form aX = b, THEN divide both sides of the equation by a to isolate X.

These rules are basically encodings of the problem solving operators in an abstract form that can apply across a range of situations. The Anderson and Thompson (1989) model shows how one can extract such problem-solving operators in the process of doing problem solving by analogy. Knowledge, once in production form, will apply much more rapidly and reliably.

The Lisp tutor is built around a cognitive model of the problem solving knowledge students are acquiring when learning Lisp programming. This cognitive model is an expert system that can solve the same problems students are asked to solve and in the same ways that students solve them. The cognitive model enables the tutor to trace each student's individual problem solving path in a process and is called model tracing, providing step-by-step help as needed. The tutor provides feedback on each problem solving action. The approach is relatively unique in the field in terms of the strong emphasis it places on the use of a real-time cognitive model in instruction. The Lisp tutor interacts with the students while they try to solve a problem on the computer. It is assumed that the student is taking an overall means-ends approach and that learning involves acquiring production rules that encode operators to use within this problem-solving organization. The tutor tries to interpret the student's problem solving in terms of the firing of a set of production rules in its cognitive model. The instruction and help it delivers to the student is determined by its interpretation of the student's problem-solving state; furthermore, its choice of subsequent problems to present to the student is determined by its interpretation of which rules the student has not mastered. One of the major technical accomplishments has been the development of a set of methods for actually diagnosing the student's behaviour and attributing segments of the problem solving behaviour to the operation of specific production rules (Anderson, 1993).

2.2. Which aspects are relevant for learning to program?

Learning to program is not easy. In an overview of what is involved, du Boulay (1989) describes five overlapping domains and potential sources of difficulty that must be mastered. These are:

(1) general orientation, what programs are for and what can be done with them;

(2) the notional machine, a model of the computer as it relates to executing programs;

(3) notation, the syntax and semantics of a particular programming language;

(4) structures, that is, schemata/plans that will be discussed in the next section;
(5) pragmatics, that is, the skills of planning, developing, testing, debugging, and so on.
None of these issues is entirely separable from the others, and much of the 'shock' of the first few encounters between the learner and the system are compounded by the student's attempt to deal with all these different kinds of difficulty at once (du Boulay, 1989, p. 284).
Rogalski and Samurçay (1990) summarise the task as follows:
Acquiring and developing knowledge about programming is a highly complex process. It involves a variety of cognitive activities, and mental representations related to program design, program understanding, modifying, debugging and documenting. Even at the level of computer literacy, it requires construction of conceptual knowledge, and the structuring of basic operations (such as loops, conditional statements, etc.) into schemata and plans. It requires developing strategies flexible enough to derive benefits from programming aids as programming environments and programming methods.

Programming	**Knowledge**	**Strategies**	**Models**
Design	(1) of planning methods, algorithm design, formal methods	(2) for planning, problem solving, designing algorithms	(3) of problem domain, notional machine
Implementation	(4) of language, libraries, environment/tools	(5) for implementing algorithms, coding, accessing knowledge	(6) of desired program
Evaluation	(7) of debugging tools and methods	(8) for testing, debugging, tracking/tracing, repair	(9) of actual program

Figure 2. Programming framework.

A structural summary is outlined in the 'programming framework' shown in figure 2. The framework gives an overview of the individual attributes of the programmer, namely their knowledge, strategies, and mental models. The framework in figure 2 summarises the relationships between issues related to programming. It should be read mainly by columns, that is, knowledge of planning methods (required to design a program), knowledge of a language (required to implement a program), knowledge of debugging tools (required to evaluate a program), and so on. In many cases we would interpret the ''cells'' of the framework as ''fuzzy'', rather than making very sharp distinctions (Robins, Rountree & Rountree, 2003).

2.3. The development from novice programmer to expert programmer.

Davies (1993) distinguishes between programming knowledge of a declarative nature, e.g. being able to state how a for loop works, and programming strategies, the way knowledge is used and applied, e.g. using a for loop appropriately in a program. There is considerable flexibility and overlap in the literature in the interpretation of the terms 'programming strategies" Davies (1993), 'knowledge structures' (Vessey, 1987), 'mental models' (Jones, 1982), 'schemas" (Detienne, 1990; Mayer, 1981), 'chunks' (Soloway &Ehrlich, 1984), 'plans' (Soloway, 1985) and 'productions ' (Anderson, 1983).
Van Merriënboer and Paas (1990) have studied the development of schemata. In their view two complementary processes may be distinguished in learning a complex cognitive skill such as computer programming. First, automation offers task-specific procedures that may directly control programming behaviour, second, schema acquisition offers cognitive structures that provide analogies in new problem situations. Learning computer programming means both learning

procedures to accomplish various goals and learning the information that is relevant to these procedures.

As a first observation, expert programmers can perform many procedures without noticeable effort because they are able to respond in a highly reflexive manner to abstract features of problems. However, their skill clearly is more than the sum of its automatic parts; when experts are confronted with new programming problems for which they have no automatic procedures available, they can rely on an enormous amount of programming knowledge that may be used by more general problem solving methods to reach a solution. Thus, besides the development of automatic procedures, the acquisition of highly structured knowledge, or schemata, plays a significant role in learning a skill like computer programming.

2.3.1. Automation

Automation leads to highly task-specific procedures that may directly control programming behaviour. In current cognitive research, such procedures are usually referred to as productions or condition-action pairs. The conditions specify various problem specifications or particular programming goals; the actions can be to embellish the problem specification, to set new sub goals, or to write or change programming code. As a result of the availability of task-specific procedures, experts can almost automatically reformulate and decompose familiar problems in sub problems that have known solutions, and they can effortlessly generate programming code to reach low-level goals, such as printing values, doing loops, or making decisions (Anderson, Farrell & Sauers, 1984).

2.3.2. Schema acquisition

Schemata can be conceptualized as cognitive structures that allow particular objects, events, or activities to be assigned to general categories. Thus, schemata provide general knowledge that can be applied to particular cases. The acquisition of several kinds of schemata is also relevant to learning elementary computer programming (Rist, 1989). For instance, a general design schema should be developed to provide abstract knowledge concerning the processes involved in generating a good design and its overall structure (e.g., Jeffries, Turner, Polson & Atwood, 1981). The design schema may then be used recursively to generate a decomposition of the problem into more and more detailed modules in a process of "stepwise refinement", which leads to a top-down, breadth first expansion of the solution. The design process continues until programming code has been identified for each of the sub problems.

Programming plans are generally considered to be a particularly important kind of schemata to acquire in elementary computer programming (Ehrlich & Soloway, 1984; Soloway, 1985). These programming plans are learned programming language templates, or stereotyped sequences of computer instructions, that form a hierarchy of generalized knowledge. High-level programming language templates (such as a general input-process-output plan) may be applied to a very wide range of programming problems, whereas medium level templates (such as a looping structure with an initialisation above the loop) and low-level templates (such as a statement to print the value of a variable) are applicable to increasingly smaller ranges of (sub)problems. Thus, programming plans provide, within the programming domain, a hierarchy of increasingly context dependent strategies that may guide a process of "templating" in the creation of solutions to posed problems.

2.3.3. The development of schemata

In the pre-novice stage of learning programming, the learner has to apply very general, weak problem solving methods such as means-ends analysis, analogy, etc. to perform the programming

task. Learning processes may either create new schemata or adjust existing schemata to make them more in tune with experience. For example, inductive processes can be described (e.g., Carbonell, 1984, 1986) that either extend or restrict the range of applicability of schemata. A more generalized schema may be produced if a set of successful solutions is available for a class of related problems, so that a schema may be created that abstracts away from the details; a more specific schema may be produced if a set of failed solutions is available for a class of related problems, so that particular conditions may be added to the schema which restrict its range of use. Research points out that such schema acquisition is a form of controlled processing, that is, it is subject to strategic control (e.g., Anderson, 1987; Proctor & Reeve, 1988). Consequently, compared to automation, which slowly develops and is mainly a function of the amount of practice, the acquisition of schemata such as programming plans may rapidly occur but requires the investment of effort, or, conscious attention and mindful abstraction from the learner.

After useful schemata have been developed, they may be used as analogies to generate behaviour in new, unfamiliar problem situations. Obviously, this will often be the case if no task-specific, automated procedures are available (i.e., triggered by cues in the current situation). The use of analogy can best be conceptualized as a kind of mapping process (e.g., Anderson & Thompson, 1989). Students may use worked examples as a kind of concrete schemata to map their new solutions; in interpreting cognitive schemata, the key to the use of the schema is interpreting it by general procedures and mapping it onto the current knowledge of the situation to create a new solution (Hesketh, Andrews & Chandler, 1989).

2.4. Known problems when learning programming by novice programmers

Which problems do novice programmers have when constructing imperative programs? Robins, Rountree & Rountree (2003) have made a review of known problems of novice programmers: Several studies that focus on novices' understanding and use of specific kinds of language feature are presented in Spohrer and Soloway (1989). Samurçay (1989) explores the concept of a variable, showing that initialisation is a complex cognitive operation with 'reading external input" better understood than assignment (see also du Boulay, 1989). Updating variables and testing variables seemed to be of roughly equivalent complexity, and are better understood than initialisation. Hoc (1989) showed that certain kinds of abstractions can lead to errors in the use of conditional tests. In a study of bugs in simple Pascal programs (which read data and perform processing in the mainline) Spohrer et al. (1989) find that bugs associated with loops and conditionals are much more common that those associated with input, output, initialisation, update, syntax/block structure, and overall planning. Soloway, Bonar, and Ehrlich (1989) study the use of loops, noting that novices prefer a ''read then process'' rather than a ''process then read'' strategy. Du Boulay (1989) notes that for loops are problematic because novices often fail to understand that ''behind the scenes'' the loop control variable is being updated (du Boulay, 1989, p. 295). Du Boulay also notes problems that can arise with the use of arrays, such as confusing an array subscript with the value stored. Kahney (1989) shows that users have a variety of (mostly incorrect) approximate models of recursion. Similarly, Kessler and Anderson (1989) find that novices are more successful at writing recursive functions after learning about iterative functions, but not vice versa. Issues relating to flow of control were found to be more difficult than other kinds of processing. Many of the points mentioned here are also addressed by Rogalski and Samurçay (1990).

Besides these language feature specific problems there are more general misconceptions. 'The notion of the system making sense of the program according to its own very rigid rules is a crucial idea for learner to grasp' (du Boulay, 1989, p. 287). In this respect anthropomorphism ('it was trying to ...', 'it thought you meant …') can be misleading. Similarly, novices know how they intend a given piece of code to be interpreted, so they tend to assume that the computer will interpret it in the same way (Spohrer & Soloway, 1989). Although prior knowledge is of course an essential starting point, there are times, when analogies applied to the new task of programming, can also be misleading. Bonar and Soloway (1989) develop this point, exploring the role of existing knowledge (e.g., of step-by-step processes), natural language, and analogies based on these domains as a source of error. For example, some novices expect, based on a natural language interpretation, that the condition in a 'while' loop applies continuously rather than being tested once per iteration.

Category	**Problem kind**	**Description**
plan composition problems	*Summarisation problem*	Only the primary function of a plan is considered, implications and secondary aspects may be ignored.
	Optimisation problem	Optimisation may be attempted inappropriately. Previous-experience problem. Prior experience may be applied inappropriately.
	Specialisation problem	Abstract plans may not be adapted to specific situations.
	Natural-language problem	Inappropriate analogies may be drawn from natural language.
	Interpretation problem	'Implicit specifications' can be left out, or 'filled in' only when appropriate plans can be easily retrieved.
	Boundary problem	When adapting a plan to a specific situation boundary points may be set inappropriately.
	Unexpected cases problem	Uncommon, unlikely, and boundary cases may not be considered.
	Cognitive load problem	Minor but significant parts of plans may be omitted, or plan interactions overlooked.
Construct based problems	*Natural-language problem*	Many programming-language constructs are named after related natural language words, and some novices become confused about the semantics of the constructs.
	Human interpreter problem	Novices know how they intend a construct to be interpreted, and so they tend to assume that the computer will be able to arrive at a similar interpretation.
	Inconsistency problem	Because novices understand how a construct works in one situation, they may assume that the construct will work in the same manner in another, slightly different situation (Spohrer & Soloway, 1989).

Figure 3. Category and description of different kinds of problems.

Spohrer and Soloway describe nine kinds of *plan composition problems* and three kind of *construct based problems (figure 3)*, some of which we have already touched upon above. *Construct-based problems* make it difficult for novices to learn the correct semantics of language constructs.
The underlying cause of the problems faced by novices is their lack of (or fragile) programming specific knowledge and strategies. While the specific problems noted above are significant, some have suggested that this lack manifests itself primarily as problems with basic planning and design. Spohrer and Soloway (1989), for example, collected data in a semester long introductory Pascal

programming course (taught at Yale University). Discussing two 'common perceptions' of bugs, the authors claim that:

> *Our empirical study leads us to argue that (1) yes, a few bug types account for a large percentage of program bugs, and (2) no, misconceptions about language constructs do not seem to be as widespread or as troublesome as is generally believed. Rather, many bugs arise as a result of plan composition problems – difficulties in putting the pieces of the program together [. . .] – and not as a result of construct-based problems, which are misconceptions about language constructs (Spohrer & Soloway, 1989, p. 401).*

2.4.1. Novice capabilities and behaviour

Robins, Rountree and Rountree (2003) say about novice capabilities and behaviour:
Novices lack the specific knowledge and skills of experts, and this perspective pervades much of the literature. Various studies are reviewed by Winslow (1996). Figure 4 show his conclusion on novices.

limited to surface knowledge (and organize knowledge based on superficial similarities);
lack detailed mental models;
fail to apply relevant knowledge;
use general problem solving strategies (rather than problem specific or programming specific strategies);
approach programming 'line by line' rather than at the level of meaningful program 'chunks' or structures;
in contrast to experts, novices spend very little time planning;
they also spend little time testing code, and tend to attempt small 'local' fixes rather than significantly reformulating programs (Linn & Dalbey, 1989);
they are frequently poor at tracing/tracking code (Perkins et al., 1989). Novices can have a poor grasp of the basic sequential nature of program execution: 'What sometimes gets forgotten is that each instruction operates in the environment created by the previous instructions' (du Boulay, 1989);
their knowledge tends to be context specific rather than general (Kurland, Pea, Clement & Mawby, 1989).

Figure 4. Conclusion of Winslow (1996) about novice capabilities and behaviour.

Some of this rather alarming list relates to aspects of knowledge, and some to strategies. Perkins and Martin (1986) note that ''knowing'' is not necessarily clear cut, and novices that appear to be lacking in certain knowledge may in fact have learned the required information (e.g., it can be elicited with hints). They characterise knowledge that a student has but fails to use as ''fragile''. Fragile knowledge may take a number of forms: missing (forgotten), inert (learned but not used), or misplaced (learned but used inappropriately). Strategies can also be fragile, with students failing to trace/track code even when aware of the process (see also Davies, 1993; Gilmore, 1990).

2.4.2. Goals and plans

Spohrer and Soloway (1989) have developed a descriptive theory of buggy novice programs that is based on the cognitively plausible, deep structure knowledge that programmers have: goals and plans. Instead of analysing a program in a construct-based approach that breaks a program down into specific syntactic constructs of the programming language. A programming plan is a schematic description of the structure of a particular piece of code, which reaches a specific goal of the program. An example is the plan to count how many times a loop has been passed. It consists of two parts: an initialization part before the loop, and an update part within the loop. In the initialization part a counting variable is set to zero ($counter = 0 ;) and in the update part this same variable is increased by one ($counter = $counter + 1 ;).

A programming plan may contain parameters, (parts of) programming constructs like a *while* loop and free variables (in this case $counter), and may refer to other plans. Goals are what must be accomplished to solve a problem (e.g. checking whether the input given by a user, is correct). Plans correspond to stereotypical sections of code that can be used to achieve the goals. Usually, a goal can be reached by more than one programming plan. Students must learn how to select the goals from the given problem text. In most cases, special terms in the problem text will cue the students to particular programming goals. Two important points about goals and plans are:
1. a goal decomposes into sub-goals, and plans organize the sub-goals of a goal;
2. there are usually many different plans for achieving the same goal.
Spohrer and Soloway (1989) identified bugs by means of a goal/plan analysis of the programs. Bugs can be identified as differences between the correct plans and the buggy implementations actually implemented by the novices (Johnson and Soloway, 1965; Spohrer, Soloway & Poppe, 1985). They categorized the found bugs and made a list of bug types. Figure 3 shows an overview of bug types.

2.5. Intelligent Tutoring Systems

To help novice programmers to overcome the above problems and to help with issues summarised in the programming framework (figure 2) an Intelligent Tutor System can be used.

2.5.1. The working of an ITS

VanLehn (2006) states that many tutoring systems can be described as having two loops. The outer loop executes once for each task, where a task usually consists of solving a complex, multi-step problem. The inner loop executes once for each step taken by the student in the solution of a task. The inner loop can give feedback and hints on each step. The inner loop can also assess the student's evolving competence and update a student model, which is used by the outer loop to select a next task that is appropriate for the student. Figure 5 shows a list with examples of tutoring systems.

Digital mathematical environment: http://www.fi.uu.nl/dwo/gr/tf/
Math bridge: http://www.math-bridge.org/
Ask-Elle: http://ideas.cs.uu.nl/ProgTutor/
Quantitative problem solving in introductory college physics: Andes.
Several, including qualitative problem solving in college physics: Autotutor.
Constructing queries to relational databases in the language SQL: SQL-Tutor.

Figure 5. Examples of tutoring systems.

There are different ways the outer loop can select a task: display a menu from which a task (or exercise) can be selected or assign tasks in a fixed sequence. Mastery learning can be implemented by keeping assigning tasks until student masters the knowledge. Macro adaptation can be implemented by maintaining detailed information about a student's knowledge, and offer tasks based on that information. Macro adaptation requires a student model. For the inner loop, we can look at five aspects: (1) Hints on the next step, (2) worked-out solution, (3) diagnosis of a student step, (4) diagnosis of a student solution and (5) knowledge assessment.
1. Hints can help students on how to proceed. A common wisdom about hints is: hint only when the student asks for it, but don't hint if the student can solve the problem without it and do hint if the student gets frustrated. Maintain a probabilistic model of which information is needed by the student. Most tutoring systems offer hints for students asking. Potential problems are help abuse and help refusal. The tutor can hint a correct step: which hasn't already been done by the student

and following the student's path to a solution. Using the terminology of the student and using the preferences of the teacher (paths, details). Next step hints are essential, in particular for ITSs with only correct/incorrect feedback. A how to hint example:

4 ∗ x = 11⇒ divide by 4 x = 2 ¾
Point: Use the procedure for solving linear equations
Teach: divide by 4
Bottom-out: divide by 4, giving x = 2 ¾

2. A Worked-out solution example:

4 ∗ (x − 1) = 7 ⇒
distribute 4 ∗ x − 4 = 7 ⇒
bring constants to right 4 ∗ x = 11⇒
divide by 4 x = 2 ¾

3. Minimal feedback can be given by: Your step is Correct/Incorrect, correct but non-optimal (longer solution, wastes resources) or unrecognizable (may be considered correct, incorrect or unrecognizable). The moment feedback is given can be immediate, delayed or on demand. 'Fading the scaffolding" by giving error-specific feedback, example:

Here is a common error: 2 + 3 ∗ x = 20 ⇒ 5 ∗ x = 20 and a possible hint sequence:
You seem to have added 2 + 3. Is that really appropriate?
You seem to have grouped 2 + 3 ∗ x as (2 + 3) ∗ x. Is that legal?
Because multiplication has a higher precedence than addition, you should have ...
You should enter 3 ∗ x = 20 − 2.

Diagnoses can also be given by error-specific feedback. Error-specific feedback can use buggy rules, supports the self-debugging process and can move from help based on a buggy rule to hints about the expected rule and can be divided into slips and potential misunderstandings. Error specific feedback should be given at the 'right' time (after making the same error twice or after correct/incorrect feedback).

4. Diagnosis of a student solution: Tutoring systems for real-time skills such as steering a ship, or fight fires review a solution after it has been submitted. Reviewing during the solving process would disrupt the activity. In non-real-time domains, delayed feedback might stimulate meta-cognitive skills. Often the form of a tutorial dialogue is used for scaffolding. What should the ITS discuss with the student? How should the ITS order these points? How deep are the discussions about these points? Can we accommodate questions (clarifications) a student might have? Aspects of the tutorial dialogues in the SQL tutor are: The number of mistakes, if any and the clause where an error has occurred. A general description of the error and more information about the error. A list can be given containing a description of every error. The correct version of the clause, where an error appeared and the ideal solution to the problem. Another point is: who controls what feedback is given when?

5. Knowledge assessment will be done for the student and the teacher, but also for the ITS itself. A coarse-grained assessment is usually computed from several measures, such as a measure of the progress and coverage (number of problems solved, number of steps correctly applied), the amount of help given (number of hint sequences, number of bottom-out hints) and competence (frequency of incorrect initial steps, time required to perform a step, number of attempts before a correct step is entered). Fine-grained assessment will be done by counting learning events (5/5 is excellent, 5/50 bad). Does a step correspond to one learning event and which event? Counting failures is possible in very structured tutors and harder in tutors with a lot of freedom.

2.5.2. Eight principles for design of tutors

Anderson and colleagues (Anderson, Boyle, Corbett & Lewis, 1990; Anderson, Boyle, Farrell & Reiser, 1987; Anderson, Conrad & Corbett, 1989) have developed an extensive and effective intelligent tutoring system for Lisp within the ACT model of learning and cognition (Anderson, 1983, 1990). Finally for a broad perspective, offered in respect to teaching Java but which could equally apply to any kind of educational situation like in this research for teaching the programming language PHP. They examined the ACT theory and extracted what they felt were eight principles for design of tutors which followed from the ACT theory and which are reviewed below.

(Details of the principles are not published in this paper but available in the MSc Thesis)

Principle 1: Represent student competence as a production set.

Principle 2: Communicate the goal structure underlying the problem solving.

Principle 3: Provide instruction in the problem solving context.

Principle 4: Promote an abstract understanding of the problem-solving knowledge.

Principle 5: Minimize Working Memory Load.

Principle 6: Provide immediate feedback on errors.

Principle 7: Adjust the grain size of instruction with learning.

Principle 8: Facilitate successive approximations to the target skill.

3. Research question

The research questions we have to answer to give recommendations for the functionality of an intelligent tutoring system for the imperative programming language PHP are:

a. What do students do when constructing PHP programs?

b. Which problems do they have when constructing PHP programs?

c. Which solution strategies do they choose?

To answer these questions we have to take a detailed look at what students do and think when they start with programming in PHP. Thus a cognitive model can be made of the problem solving knowledge students acquire when learning PHP programming.

4. Method

How can we take a detailed look at what students do and think when they start with programming in PHP? Computer-mediated activities can be recorded automatically. Such logs can be comprehensive, precise and accurate. From automated logs it is possible to reconstruct with detail and accuracy the conduct of the task and can be divided into many subjects. For example: to analyse sequences of actions, association between actions and errors or actions and outcomes, and time spend on different components. The disadvantage is that, although they record precisely what people do when they interact with the system, the recordings of the screen offer no direct information about what the people intended to do, or where they looked, or what they did when they were not interacting with the system. Most of these disadvantages can be overcome by letting people think aloud and record this and tape a video. The logs and recordings produce a lot of precise and detailed data. The planning of collection and analyses of that stream requires careful reasoning about how to interpret research questions and how to filter and to manipulate those data relevantly (Fincher & Petre, 2004).

Particular knowledge about the problem-solving process can be acquired by using think aloud protocols (van Someren, Barnard & Sandberg, 1994). They made a simple model of the human cognitive system as shown in figure 6.

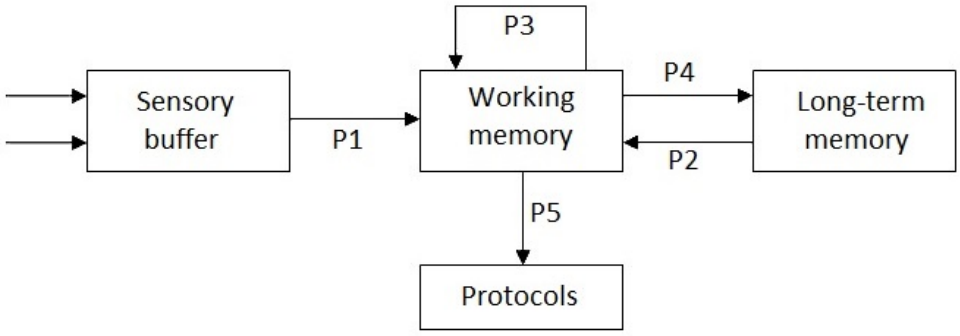

Figure 6. Model of the human cognitive system.

Long-Term Memory (LTM) is the part where knowledge is stored more or less permanently. It takes some time to store information there and it can be retrieved later on to be used again. At the other end we find the sensory system that transforms information from the environment into an internal form. Working Memory (WM) is the part where the currently 'active' information resides. In this model there are five processes:

1. Perception: Information flows from the sensory buffer into the working memory.
2. Retrieval: Information is retrieved from long-term memory into the working memory. It still exists in long-term memory but is activated into the working memory.
3. Construction: New information is constructed from other information in the working memory. For example, when solving a physics problem, someone may note that 'slowly moved piston' may in general refer to 'adiabatic process' and the resulting new association between these concepts is stored as a new object in the working memory.
4. Storage: Stores information from the working memory into long-term memory.
5. Verbalization: Information that is active in the working memory is put into words. The output of this process is the spoken protocol.

The model has several important implications for the meaning of verbal reports. One important point is that the information that can be verbalized is the content of the working memory. This means that the content of long-term memory (the general knowledge) cannot be verbalized (unless it is somehow retrieved rather than used), nor can the cognitive architecture, the machinery, that applies the knowledge be verbalized. About these aspects only indirect knowledge is available.

During computer programming both products of problem-solving: in the form of answers to exam questions and solutions produced by students during practical programming work can be logged. The recorded think aloud protocols, video and the logged screen data will be transcribed, segmented and categorized by a coding scheme (van Someren, Barnard & Sandberg, 1994). The known programming problems mentioned above will be part of the coding scheme. Other categories will be the goals and the plans/steps/operators (the typed PHP code) used to solve the programming problems.

4.1. Setting, context and participants

The data were collected from eight students (7 male, 1 female) of the 11th grade secondary school of the pre-university stream (16-18 years old) during the computer science lessons in spring 2012 at the

Maaslandcollege in Oss the Netherlands. It was their second year of computer science. They learned about computer hardware, the binary number system, logical circuits and the basic principles of how a processor works. They worked with MSWLogo for about five lessons. MSWlogo is a program that lets a turtle make figures by giving it commands. They also learned about HTML and a little bit of CSS. They built a website consisting of five pages, and used the most common structures of HTML. The students started to practice programming with Programming Structure Diagrams (PSD) or Nassi-Shneiderman diagrams (Nassi & Shneiderman, 1973). Those diagrams are a graphical representation of program structures. The main purpose of a Nassi-Shneiderman diagram is to create a graphical and logical structure for the development of structured programs. The basic principles of programming taught are: variables, sequence, selection, iteration, arrays, strings and functions. That took about five lessons and taught the basics of imperative programming. Then, the PHP programming language was introduced together with the theory of languages for the web and the client server theory. Originally PHP stood for 'Personal Home Page', now it stands for 'PHP Hypertext Preprocessor'. PHP is a general-purpose scripting language that is especially suited to server-side web development where PHP generally runs on a web server. Any PHP code in a requested file is executed by the PHP processor module runtime which generates the resulting web page, usually to create dynamic web page content (e.g., to process forms filled out by the user). PHP code is used in combination with the Hypertext Markup Language ((X)HTML) and is not visible to the user on the client side. PHP syntax is similar to most high level languages that follow the C style syntax. PHP is open source software and available free of charge. Other scripting languages used for the same purpose are Java Server Pages (JSP), ColdFusion, Active Server Pages (ASP), Perl, Python and Ruby. The students used the PHP book: 'PHP keuze module programmeren', written by Peter Kassenaar (2005). The PHP programs would be executed by a web server software: 'USBWebserver' (version 7.0). The editor they used to write the PHP programs is 'notepad++' (version 5.6.7). This editor had some support for the programming language PHP, like different colours for variables and PHP reserved words and supports finding matching brackets. The editor had no support for the syntax of PHP. Except for one student, whose logs have not been used, the participants had no experience with PHP or other programming languages.

4.2. Data collection

The programming subjects that were looked at in detail are: variables, selection and iteration. The student activities were registered by logging students' activities during programming and letting them think aloud. The logging was done by the 'Camstudio' software (version 2.0). The screen and the voice were recorded during the sessions. Three different sessions were logged (figure 7). The first session was taken during a normal lesson and students had to get used to the situation and made some fun in the beginning with each other, later on they became more serious. Figures 8 and 9 show the exercises used during the sessions. The second session was taken during a test situation. During the test, the students did not 'think aloud' a lot, because it disturbed their concentration and they found it irritating to hear the other students think aloud. During the test-session they were working more seriously than in the first session. From each of those sessions a video was recorded of the entire class with one student in front when they were constructing the PHP programs. The third session consisted of in depth interviews with three students who did the test less well. The same exercises were used as the ones that were used during the test. The recorded screen information and voices were transcribed to a text document (an example is given in appendix 1 of the MSc-thesis).

Figure 7 presents the overview of logged and transcribed sessions. The meaning of codes are: V: logged session; X: not logged; F: failed logging; N: Not interviewed; T: transcribed logging with reference to transcribed file name.
Reference example [L6.1.2] stands for: student number: L6, session number: 1, exercise number: 2

Student and Student number	**session[1] 09-03-2012**	**session[2] 20-03-2012**	**session[3] 04-2012**
Aron [L1]	T[L1.1]	F	N
Ben [L2]	V T	[L2.2]	N
Claire [L3]	T [L3.1]	X	T[L3.3]
George [L4]	X	V	T [L4.3]
John L5]	T [L5.1]	V [L5.2]	N
Peter [L6]	T [L6.1]	X	T [L6.3]
Edward [L7]	V	T [L7.2]	N
Dave [L8]	F	V	N

Figure 7. Overview of logged sessions and transcribed sessions.

4.2.1 Exercises used during the recorded sessions

Figure 8 shows the exercises in session 1 and figure 9 shows the exercises in the session 2 and 3.

Exercise 1: 'the sum problem using a while loop' *Create a program using HTML and PHP, that (with a while loop) determines what the outcome is when you add the numbers 1, 2, 3 …. until a number that you set in the code in advance, and then shows the outcome on the screen. (For example, if the number has the value 5, the calculation is 1 + 2 + 3 + 4 + 5 and the result is 15, that will be printed on the screen). Show also how the calculation is done. See the example below:* *The total of 1 to 5 is: 1 + 2 + 3 + 4 + 5 = 15*
Exercise 2: 'the sum problem using a for loop' *The same exercise as exercise 1, but now you have to use a 'for-loop'*

Figure 8. Session[1] PHP exercises using loops (09-03-2012).

Exercise 1: 'The alcohol problem' *Create a program that, depending on the age, puts on the screen:* *"you're younger than 16: no alcohol!"* *"You're older than 16 but not yet 18, you may drink light alcoholic beverages."* *"You're older than 18, you may also drink spirits."* *Test your program for every age category.*
Exercise 2: 'The die problem' *Let the computer throw a die (using the function rand(1,6)) 50 times and count how many times six is thrown and show this on the screen. Print the outcomes on the screen and show an update of how many times six has been thrown till then.*

Figure 9. Session[2] exercises of test situation(20-03-2012) and Session[3] depth interviews(04-2012).

5 Analysis of the data

The recordings and logs are analysed according the following questions: What do students do when constructing PHP programs? Which problems do they have when constructing PHP programs and which solution strategies do they choose?

5.1 What do students do when constructing programs?

The logs of the data and the video recordings show that students in general: read the programming exercise, looked in their PHP-book for an example of the programming construct they wanted (or had) to use. According to the goal the students have, they make a piece of programming code, test it, interpret the results and correct the code and/or add code to reach their goal to get to the next state, most of the time, closer to the solution of the problem. In the example in appendix 1, we see that the student Aaron [L1.1] read the problem, typed the standard HTML tags, searched for an example of the while loop in his PHP book, converted the example to the situation asked by the exercise and typed the PHP code, tested his solution, interpreted the results of the test, changed the PHP code to get the results closer to the end-result etc. until he reached the solution of the problem. In the next section we take a closer look at the code the students develop and the problems they encountered.

5.2 Which problems do novice programmers have when constructing PHP programs

The text files with the logged data are analysed by looking at the students goals and plans and the programming problems they encounter. First, a possible solution of the exercise is given then the goals and plans logged by the students. The solutions of the students are translated from Dutch. The names of the variables used by the students are transformed to variable names used in the example solution. Syntax problems are left out of the analysis. The known problems the students encountered are categorised in the bug types given in the 'known problems' section. The problematic PHP statements are given in italics in the 'plans/steps/operators' column.

The first programming exercise which is analysed ('the sum problem') is a programming problem what states which programming construct the students have to use: the while loop. In the next exercise, the same problem must be solved by using a for loop. In 'the alcohol problem' exercise the selection construct is used. These exercises can be solved by using one programming construct. In 'the die problem' the students have to invent which programming constructs they are going to use and they have to combine two programming constructs.

Details of exercises (5.2. to 5.4.)are not published in this paper but available in the Master Thesis:

5.2.1. Exercise: 'the sum problem using a while loop'

5.2.2. Exercise: 'The sum problem, using a for loop'

5.2.3. Exercise: 'The alcohol problem'

5.2.4. Exercise: 'The die problem'

5.2.5. Evaluation of the programming problems and which problems are specific for PHP

5.4. Which solution strategies do the students choose?

Strategies can be ordered according to the main steps in the problem solving process - namely, (a) analyzing the problem, (b) designing an algorithmic solution, (c) implementing the solution, and (d) testing, debugging, and maintaining the program (Krammer, van Merriënboer & Maaswinkel, 1994). See also the strategies column of the programming framework in figure 2. Programming strategies, the way knowledge is used and applied, to find out by which programming constructs the programming problem can be solved and e.g. using a *for* loop appropriately in a program (Davies,1993). Which solution strategies do the students choose? The students have goals and they try to reach those goals by developing lines of programming code. The questions the students formulate during the think aloud sessions to determine their strategies are:

* How can I make it do something 50 times? [L8.3.2]
* How can I make a value (or variable) and add 1 to it? [L6.3.2]

* How can I make it stop? [L6.3.2]
* When it is 50, it has to stop, how can I do that? [L6.3.2]
* How can I put this value on the screen? [L8.3.2]
* How can I see the "in between" value of this (variable/value)? [L8.1.1]
* Show me an example of this command? [L8.1.1]
* What does this mean (e.g. $i++)? [L6.3.2]
* How can I let it do nothing (*else construct*)? [L6.3.2]
* It doesn't work, what do I wrong? [L7.2.2]

5.4.1. Programming development behavior and feedback

When feedback is given immediately during problem solving, students do not have to check their own work because the ITS will point out their errors to them. They also do not have to deal with getting lost and starting over, because they can ask the ITS for a hint when they feel lost. Thus, certain meta-cognitive skills may not be exercised during problem solving because the ITS makes them unnecessary. When students get delayed feedback only, they must practice those meta-cognitive skills themselves. That should enable them to solve problems when the ITS is no longer present. As mentioned earlier, the outer loop can 'fade the scaffolding' by initially assigning tasks with immediate feedback and later assigning tasks with delayed feedback (van Lehn, 2006). Studying the students' programming code, we see that students correct their own code after some time. The moment a student tests his programming code, can be a good moment for feedback. Testing the code can be seen as a form of asking for feedback. Figure 10 shows how many times a student tests his program during the solution of 'the die problem'.

Student	With syntax errors	Without syntax errors	Total
Edward [L5.2]	1	2	3
George [L6.3]	7	4	11
John [L7,2]	7	7	14
Peter [L8,3]	3	11	14

Figure 10. Number of times a student tests his program during the solution of 'the die problem'.

Figure 10 shows that the more often the student tests his program, the more problems he has with the syntax and the solution strategy. The more often he tests his program the more states he needs to solve the problem. The logs show that some students look to their programming code for a short time and make a change of their code and test the program again. They do not take the time, or do not have the capabilities to understand what the program is doing exactly. Edward tested his program only three times. He mastered the programming constructs and developed a successful strategy to solve "the die problem".

6. Recommendations for the functionality of an intelligent tutor system for PHP programming

The recommendations for an ITS for the programming language PHP consist of production rules to inform the cognitive model, 'Frequently Asked Questions' together with examples to support the instruction and some general aspects. Finally we show an interactive session with a hypothetical ITS for PHP in which our recommendations are implemented.

6.1. General aspects

Novice programmers make a lot of syntax errors. In logged data results 18 errors were related to syntax errors of 42 errors in total. An advanced syntax checker can be of help to detect the syntax

errors in the PHP programming code. There are already more advanced syntax checkers available. That functionality can be added in an ITS for PHP. Students tried to evaluate their program code and the variables they used and tried to show the content of the variables on the screen. It was difficult for them to do this. A report on the screen of all the used variables and the executed programming code, generated by the ITS, can help the students to evaluate their programs.

6.2. Production rules for the programming structures of PHP

An ITS interacts with the students while they try to solve a problem on the computer. It is assumed that the student is taking an overall means-ends approach and that learning involves acquiring production rules that encode operators that can be used within a problem-solving organization. The ITS tries to interpret the student's problem solving in terms of the firing of a set of production rules in its cognitive model. The instruction and the help provided to the student is determined by the interpretation of the student's problem-solving state; furthermore, its choice of subsequent problems to present to the student is determined by the interpretation of which rules the student has not mastered (Anderson, 1993).

As already mentioned in one of the eight design principles above (principle 7), declarative instruction using hypertext facilities that can be accessed in parallel with the ITS. The content of this instruction is informed by the production rules that are to be learned. The instruction tries to provide examples that illustrate the rules and annotate those examples with comments that will highlight the significant aspects of the rules. A general principle in the approach to instruction is to be minimalist and not say more than is needed (Reder & Anderson, 1980; Reder, Charney & Morgan, 1986; Brusilovsky & Millan, 2007).

According to the theory of ACT and the means-end analysis and what we have seen in the logs is that every student takes an example of the programming structure he wants to use. The use of the programming structures *if-else-elseif*, *while*, *for*, variables and the *echo* statement can be divided into steps. The production rules for the programming structures of PHP are combined with an example of the programming structure it concerns. The terminology of the production rules is derived from the terminology the students used during the logged sessions. An example of how this can be done is given in sub paragraphs.

6.2.1. If you want to write something on the screen then you can use the echo statement.
6.2.2a. If you want to make the output easier to read then use spaces and newlines (
).
6.2.2b. If you want to know the content of a variable, you can put it on the screen.
6.2.3. If you want to use a value you can use a variable.
6.2.4. If something has to be done more than once then you can use a loop.
6.2.5. If you want to use a while loop then take the following steps.
6.2.6. If you want to use a for loop then take the following steps.
(Remark: subparagraphs 6.2.1. to 6.2.6. are not included in this article. See details in the MSc-Thesis).

6.3. Frequently Asked Questions.

The questions the students formulate during the logged sessions to determine their strategies, can lead to a list of 'Frequently Asked Questions' (FAQ) of the ITS for a specific exercise, or general questions which are available for all exercises. Those questions help the student to master the problem space of the programming language and to develop the students' strategies. An interesting fact, derived from the logged data, is that the novice programmer does not think in the problem

space of the programming language, they do not think in programming constructs. The ITS can help to make a bridge between the language of thinking of the novice programmer to the problem space of the programming language. Figure 11 shows the questions the students asked themselves during the problem solving of the programming exercises and a possible reaction of the ITS.

Question asked by the students	**Possible reaction of the ITS**
How can I make it do something 50 times?	The ITS gives an explanation of the different kinds of loops the programmer can use.
How can I make a value (or variable) and add 1 to it.	The ITS gives an explanation and examples of variables.
How can I make it stop?	The ITS explains the use of loops, or the loop the student is using.
When it is 50, it has to stop, how can I do that?	The ITS gives an explanation about loops in general, or about the loop the student is using.
How can I put this value on the screen?	The ITS gives an explanation and examples about the 'echo' statement.
How can I see the 'in between' value of this (variable/value)?	The ITS gives an explanation and examples about the 'echo' statement.
Show me an example of this command.	An example of the command that has been selected by the student is given. The example given is related to the programming exercise the student is solving.
What does this mean (e.g. $i++)?	An explanation of the selected code (in this case: $i++) is given by the ITS.
How can I let it do nothing (else construct)?	The ITS gives an explanation and examples about the different forms of the selection construct.
It doesn't work, what did I do wrong?	The ITS can give a report on the screen of all the used variables and the executed programming code.

Figure 11. Questions of students and possible reaction of the ITS.

6.4. Interactive session with a hypothetical ITS for PHP with our recommendations implemented.
(Remark of editor: subparagraph6.4. is not included in this article. See for details the MSc-Thesis).

7. Discussion and Conclusion

The aim of this research was to give recommendations for an ITS for the program language PHP. To give recommendations we studied novice programmers on what they did when constructing PHP programs, which problems they had and which solution strategies they chose.

We took a detailed look at what students did and thought when they started with programming in PHP. We logged the screen output and let the students think aloud when they were solving programming problems and constructing PHP programming code. We recorded a video of one of the students in front to see what the students did. We looked in particular at the programming structures if-else-elseif, while, for, variables and the echo statement.

The logs and recorded videos are transcribed to text files. The results show that the students read the programming exercise, look in their PHP-book for an example of the programming construct they want (or have) to use. Depending on their goal the students construct a piece of programming code, test it, interpret the results and correct the code to reach their goal or add code to become closer to

the solution of the problem. Most of the programming problems they encountered were known issues and documented in the literature (Robins, Rountree & Rountree, 2003).
Issues specific for the programming language PHP, in combination with the USBwebserver software, are: the 'output problem', 'PHP code is on the screen problem', the 'blank screen problem' and the 'weak syntax checker problem'. The 'weak syntax checker problem' can be solved by a more sophisticated syntax checker in the ITS. To help the student to evaluate his program the ITS can show the content of all the variables and the programming code well on the screen.
The questions the students formulated, during the logged sessions to determine their strategies, lead to a list of 'Frequently Asked Questions' (FAQ) which are to be answered by the ITS. These questions can help the student to master the problem space of the programming language and to develop the students' strategies. The questions the students formulated during the think aloud sessions show that they do not think in the programming constructs of PHP. The novice programmer does not think in the problem space of the programming language. The ITS can help to make a bridge between the language of thinking of the novice programmer to the problem space of the programming language. The production rules for the programming structures of PHP, to support the cognitive model, are combined with an example of the programming structure it involves. These examples are added to support the instruction. The terminology of the production rules is derived from the terminology the students use during the logged sessions. Finally we showed an interactive session with a hypothetical ITS for PHP in which our recommendations are implemented.
The students of this research were prepared for programming by using graphical 'turtle' software (MSWlogo) and programming with programming structure diagrams (PSD's). This could have influenced the results. They were not completely novice programmers. This could explain why the students did not have many problems with simple programming exercises. Another aspect that could have influenced the results is that thinking aloud can influence the problem solving of the students (Ericsson & Simon, 1993). During the test session, the students did not 'think aloud' a lot, because it disturbed their concentration and they found it irritating to hear the other students thinking aloud. We hope that the recommendations of this research can inspire designers of ITSs to implement the recommended functionality and by doing so helping novice programmers with the difficult and challenging task of learning how to program in PHP. Further research can be done to investigate whether our recommendations help novice programmers to learn programming in PHP.

Literature

More than 80 registered references are not included in this article (see MSc-Thesis).

Appendix 1

Example logs of data and partial transcripts are not included in this article (see MSc-Thesis).

Wilt u reageren op dit artikel en/of presentatie? Neem dan contact op met:
Gerrie van Leeuwen; docent Informatica; voortgezet onderwijs
gwmvanleeuwen@hotmail.com