Recommendations for the functionality of an intelligent tutor system for the programming language PHP

Master science education and communication of computer science / informatics, University of Utrecht, the Netherlands (CROHO label: 68076)

MSc Thesis: Gerrie van Leeuwen

Supervisor: Prof. Dr. Johan Jeuring
Internal coordinator: Dr. Arthur Bakker

Date: 05 December 2012

Abstract

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 can not 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 Merrienboer & Paas, 1990). Next, the known problems (Spohrer& Soloway, 1989) and the capabilities and behavior 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).

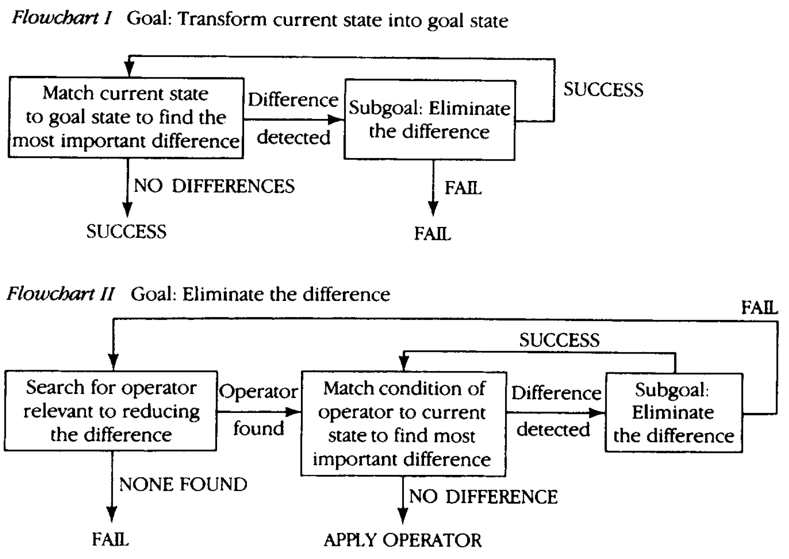
2.1 Basic concepts in problem solving and learning

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 number of 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. An flowchart of the Means-Ends Analysis is given in figure 1.

Figure 1. Application of the Means-Ends Analysis

Application of Means–Ends Analysis


Flowchart I Goal: Transform current state into goal state
Match current state to goal state to find the most important difference
Difference detected
Subgoal: Eliminate the difference
SUCCESS
NO DIFFERENCES
SUCCESS
FAIL
FAIL
Flowchart II Goal: Eliminate the difference
FAIL
SUCCESS
Search for operator relevant to reducing the difference
Operator found
Match condition of operator to current state to find most important difference
Difference detected
Subgoal: Eliminate the difference
NONE FOUND
FAIL
NO DIFFERENCE
APPLY OPERATOR

Note. Flowchart I breaks a problem down into a set of differences and tries to eliminate each. Flowchart II searches for an operator relevant to eliminating a difference.

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 are 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 behavior and attributing segments of the problem-solving behavior 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.

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.

Figure 2. programming framework

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

The framework in figure 2 summarises the relationships between a number of issues relating 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 behavior, 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 behavior. 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 behavior 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.

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).

Spohrer and Soloway describe nine kinds of *plan composition problems* and three kind of *Construct-based* problems, some of which we have already touched upon above:
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. These are problems that make it difficult for novices to learn the correct semantics of language constructs.
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).

2.4.1 Novice capabilities and behavior

Robins, Rountree and Rountree (2003) say about novice capabilities and behavior:
Novices lack the specific knowledge and skills of experts, and this perspective pervades much of the literature. Various studies as reviewed by Winslow (1996) he concludes that novices are:

- 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).

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 analyzing 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 then 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. An overview of bug types is given above.

2.5 Intelligent Tutoring Systems

To help novice programmers to overcome the above problems and to help them with the issues summarised in the programming framework of figure 2 an Intelligent Tutor System can be used. How does an ITS work?

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. Examples of tutoring systems are:
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.

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:

> $4 * x = 11 \Rightarrow$ divide by 4 $x = 2\ 3/4$
> Point: Use the procedure for solving linear equations
> Teach: divide by 4
> Bottom-out: divide by 4, giving $x = 2\ 3/4$

2. A Worked-out solution example:

> $4 * (x - 1) = 7 \Rightarrow$
> distribute $4 * x - 4 = 7 \Rightarrow$
> bring constants to right $4 * x = 11 \Rightarrow$
> divide by 4 $x = 2\ 3/4$

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: $2 + 3 * x = 20 \Rightarrow 5 * x = 20$ 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:

Principle 1: Represent student competence as a production set. The fundamental insight is that the tutoring enterprise should be informed by an accurate model of the target skill. The cognitive model allows us to set appropriate curriculum objectives and to properly interpret the actions of the student. Decomposing a skill into components and organizing instructions according to the componential analysis.

Principle 2: Communicate the goal structure underlying the problem solving. One of the enduring assumptions of the ACT theory has been that solving a problem involves decomposing that problem into a set of goals and sub goals. So the reasonable assumption was that exposing and communicating such goals should be an instructional objective. Collins and Brown adapted an approach that has been called reification (Brown, 1985; Collins & Brown, 1987). They attempted to develop interfaces that made explicit the goal structures which were only implicit in the instruction.

Principle 3: Provide instruction in the problem solving context. This principle was based on the research showing the context-specificity of learning (e.g., Anderson, 1990; Ch. 7). The difficulty with this principle is that there is not a detailed theoretical interpretation of why it is true and so it is a little hard to know how to apply it in detail. Does this mean provide instruction in the same class session as the tutor is used? before each problem? in the midst of each problem? As it has evolved in their applications this has come to mean providing instruction between each new section in the tutor (a section is where new production rules are introduced) allowing the student to refer back to this instruction in the course of problem solving. They have experimented with placing instruction at the precise point where it is needed in a problem but students find this interferes with their problem solving.

Principle 4: Promote an abstract understanding of the problem-solving knowledge. This principle was motivated by the observation that students will often develop overly specific knowledge from particular problem-solving examples. In terms of production rules this has meant that the conditions on the rules were not sufficiently general. While the problem is undoubtedly real this principle provides no guidance for how it is to be achieved. In practice they tried to reinforce the correct abstractions in the language of the help and error messages.

Principle 5: Minimize Working Memory Load. This principle was motivated by the fact that learning a new production rule in ACT requires that all the relevant information (relevant to the condition and action of the to-be-learned production) be simultaneously active in memory. Keeping other information active could potentially interfere with learning the target information. Sweller (1988) has shown that a high working-memory load interferes with learning. This principle means minimizing presentation and processing of information not relevant to the target productions. This includes minimizing presentation of instruction while problem solving since processing this instruction poses another working memory load.

This also implies that one should try to provide instruction on specific components only when other components of the skill have already been relatively well mastered. This leads to a curriculum design in which only a few new things are taught at a time. This could be viewed as being at odds with the current approaches such as cognitive apprenticeship or anchored instruction, which advocate teaching component skills in the context of complex, real-world problems. However, this approach does not deny the value of learning in such context but rather argues that students should gradually acquire the skills required to deal with this complexity rather than having to acquire them all at once.

Principle 6: Provide immediate feedback on errors. This clearly has been the most controversial of their tutoring principles. The ACT* theory claimed that new productions were created from records of problem-solving traces. Therefore, the longer one waited until an error was corrected the longer the span of problem solving over which the student would have to integrate to create a production. The current ACT-R theory claims that one learns from problem-solving products. Thus, the learner examines the resulting solution (code, proof, algebraic derivation) and builds productions from that. Thus, it does not matter whether all the critical steps occur together in time or not, only that they be represented in the final solution. *Thus, the principal theoretical justification for immediate feedback no longer exists in ACT-R.* Immediate feedback can be beneficial in cutting down on time spent in error states and making it easier to interpret the student's problem solving.

Principle 7: Adjust the grain size of instruction with learning. This principle was motivated by the composition learning operator in ACT which claimed that single productions would be composed into larger productions which did in one cognitive step what had been done in many steps. Thus, it seemed reasonable to design the interface so that one could process the student's problem solving in ever larger units of analysis.

Declarative Instruction
Anderson and colleagues have had success to give declarative instruction using hypertext facilities that can be accessed in parallel with the tutor. The content of this instruction is informed by the production rules that are to be learned in the upcoming section. 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 their approach to instruction is to be minimalist and not say more than is needed. This sensible approach tends not to be followed in most textbooks but is well supported by research (Reder & Anderson, 1980; Reder, Charney & Morgan, 1986).
While the tutor-external instruction is important, of more concern to the tutor development system is the declarative instruction delivered from within the tutor. This is of two kinds:

(1) Error Messages. When the student makes an error one can present a message that attempts to tell the student something useful about that error. This requires writing buggy productions and attaching instruction from these productions. In general they do not attempt to provide any deep diagnosis of the cognitive origins of the error. Rather simply try to explain why it is an error.

(2) Help Messages. At various points in time the student can request help or be judged in need of help and a help message can be generated. These are generated from templates associated with the correct productions which would have fired at that point.

Principle 8: Facilitate successive approximations to the target skill: Frequently, when students are initially trying to perform a skill, they cannot perform all the steps. They had the tutor fill in the missing steps. The expectation was that with repeated practice this division of labor between student and tutor would change with the student providing more and more of the work until the tutor was completely in the background. In practice this successive approximation has frequently worked quite well. This principle seems quite analogous to "fading" in the cognitive apprenticeship terminology (Collins, Brown & Newman, 1990).

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?

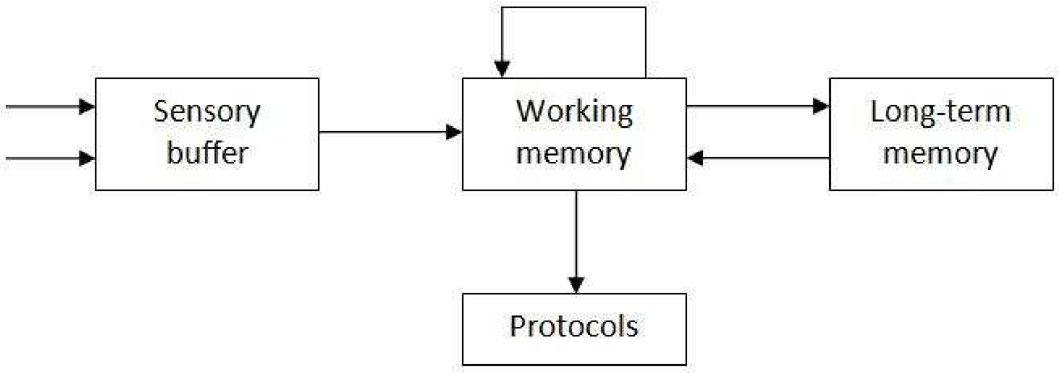
To answer these questions we have to take a detailed look at what students do and think when they start with programming in PHP. Thus a cognitive model can be made of the problem solving knowledge students acquire when learning PHP programming.

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 to tape a video. The logs and recordings produce a flood of precise and detailed data. The planning of the 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 3.

Figure 3. 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 was 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 was recorded during the sessions. Three different sessions were logged: The first session was taken during a normal lesson. During this first session they had to get used to the situation and made some fun in the beginning with each other, later on they became more serious. The exercises used during the sessions are given below. 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.

Table 1. Overview of logged sessions and transcribed sessions

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]
Gerorge [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

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

4.2.1 Exercises used during the recorded sessions

SESSION [1]: PHP exercises using loops (09-03-2012)

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".

SESSION [2] exercises of test situation (20-03-2012) and SESSION [3] in depth interviews (04-2012)

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.

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 (see tables 2,4,5 and 6) then the goals and plans logged by the students (see tables 3 and 7). 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 (see table 2). In the next exercise, the same problem must be solved by using a *for* loop (see table 4). In "the alcohol problem" exercise the selection construct is used (see table 5). 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 (see table 7).

5.2.1 Exercise: ***"the sum problem using a while loop"***

Table 2. A possible solution in PHP of "the sum problem, using a while loop"

```
<html>
<head>
<title> The sum problem, using a while loop </title>
</head>
<?php
$i = 1;
$sum = 0;
$number = 5;
echo " The total of 1 to $number is:<br> ";
while ($i < $number){
      echo "$i+";
      $sum = $sum + $i;
      $i = $i + 1;
}
echo "$number = ";
$sum = $sum + $number;
echo “$sum”;
?>
</html>
```

Table 3. Survey of the goals, plans and problems the students encounter when they develop a program for the “the sum problem, using a while loop”.

Goals	Plans/steps/operators	Problems
Type the general HTML tags in which the PHP code is embedded, and save the PHP program in the “root” directory of the USBWebserver software on the USB-stick.	<html> <head> <title> exercise 1 </title> <?php ?> </head> </html>	

Construct a while loop running from 1 till a given number, using an example of the PHP book	$i = 1;
	While ($i < $number){
	
	While ($i <= $number){
	
	While ($sum < $end_result){
		*Boundary problems:
		
		- < and <= problems)[L1.1]
		
		- choosing the wrong variables: the running total ($sum) or using the end result (e.g. 15 (the sum from 1 till 5) in stead of the number (5, $number) [L7.1]

Add the number of values	$number = 5;	
Print the added values on the screen	Echo “+ $i”; *Echo “$i + 1”;* *Echo “$i +”;*	*Output problems: - making the output as asked in the problem question [all students, all problems]
Update the sum of the added values, starting with the value 0	$sum = 0; $sum = $sum + $i; *$sum = 0 + $i;* *$sum = $i + 1;* *$sum = $sum + $i;*	*Updating variable problems, *Initialisation problem: -using a variable without giving it an initial value [L1.1] [L7.1] [L8.1]
Increment the loop counter, this is also the next value to be added	$i = $i + 1; *$sum = $sum + 1;* *$i = $sum + $i;* *$sum = $i + 1;*	*Updating variable problems: - increment the wrong variable or increment the wrong way [L7.1]

Update the summation result and put the sum on the screen	Echo = "= $number ";
	$sum = $sum + $number;
	Echo "= $sum";
	
	Echo "= $sum";
		*Boundary problem:
		
		- forget to add the last number ($number) [L1.1]

5.2.2 Exercise: *"The sum problem, using a for loop"*

"The sum problem, using a for loop" is about the same summation as the "the sum problem, using a while loop" but with this task the students have to use a *for* loop. All the students took the solution of the "The sum problem, using a while loop" and took an example of the for loop from the PHP book and replaced the *while* construct by the *for* construct and removed the counter update "$i = $i + 1;". No problems were detected except some syntax errors. Only Peter [L8.2] asks: "You don't have to use "$i = $i + 1" do you?"

Table 4. A possible solution in PHP of "The sum problem, using a for loop"

```
<html>
<head>
<title> The sum problem using a for loop </title>
<?php
$i = 1;
$sum = 0;
$number = 5;
echo " The total of 1 to $number is:<br> ";
for($i=1;$i<$number;$i++){
     echo "$i+";
     $sum = $sum + $i;
}
echo "$number = ";
$sum = $sum + $number;
echo "$sum";
?>
</head>
</html>
```

5.2.3 Exercise: *"The alcohol problem"*

To solve this programming problem the students used an example of the *if-else-elseif* construct in their PHP book. Apart of some syntax errors, none of the logged students had a problem with this task.

Table 5. A possible solution of "the alcohol problem".

```
<html>
<head>
<title> The alcohol problem </title>
<?php
$age = 15;
if ($age < 16)
echo "you're younger than 16: no alcohol!";
elseif ($age >= 18)
echo "You're older than 18, you may drink also spirits.";
else
echo "You're older than 16 but not yet 18, you may drink light
```

```
alcoholic beverages.";
?>
</head>
</html>
```

5.2.4 Exercise: *"The die problem"*

With *"the die problem"* the students have to decide by themselves, which programming constructs they use (iteration: *while* loop or *for* loop, in combination with an *if* construct).

Table 6. A possible solution of "the die problem".

```
<html>
<head>
<title> The die problem </title>
<?php
$i=1;
$count_six=0;
While ($i<=50){
      $number = rand(1,6);
      echo "$i the thrown number is $number <br>";
      if ($number==6){
            $count_six++;
            echo "6 has been thrown $count_six times.<br>";
      }
      $i = $i + 1;
}
Echo "In total $count_six times 6 has been thrown";
?>
</head>
</html>
```

Table 7. Overview of the goals, plans and problems the students have when they develop a program for "the die problem" exercise.

Goals	Plans/steps/operators	Problems

Make the computer do something 50 times: by constructing a while loop	$i = 1;
	While ($i <=50){
	
	While ($i < 50){
	
	If ($number = 50) {
		*Boundary problems:
		
		- < and <= problems[L2.2] [L4.3]
		*Natural language problem:
		- using *if* as loop instead of *while*[L4.3]

or by constructing a for loop	For ($i=1;$i<=50;$i++){	
or by constructing a do .. while loop	Do (… } while $i<50); *Do (…* *$i = $i + 1;* *} while $a<50);* *If ($i = 50)*	*Boundary problems: - < and <= problems .[L5.2] - using the loop variable ($a) from the example and increment the own chosen variable $i [L5.2] * Natural language problem - trying to make a loop with the *if* construct[L6.3]

Make a die (using the random function of PHP rand(1,6))	$min=1; $max=6 $die=RAND($min,$max); $die=rand(1,6);	
Print the thrown value on the screen	Echo ”$die “;	*Output problems: -making the output organized, with spaces and new lines [all students, all problems]
Print how many times the die is thrown, on the screen	Echo “$i “;	*Output problems: -making the output organized, with spaces and new lines [all students, all problems]
Count how many times six is thrown	If ($die == 6) $count_six = $count_six + 1; $count_six++; $number = 6; if ($die == $number); *If ($die == 6)* *$count_six = $count_six + 1;* *else* *$i=$i+1;* *else {* *$goed = false;* *$i++;* *}* *If ($die == 6)* *$count_six = $count_six + 1;* *Else* *$count_six = $count_six;* *if($die=6){* *if (rand (1,6) = 6)* *$number=6;* *if ($die == $number){* *$good = true;*	*Cognitive load problem: - not count how many times six is thrown [L4.3] [L6.3] [L6.3] *Inconsistency problem, construct based problem: *The empty else problem: -try to let the else does nothing by using an empty line [L2.2] -the statement after the else will be executed when the if condition is not true, this is not noticed by the student [L2.2] -loop counter $i will only be incremented when not six is thrown (loop counter only incremented in else condition) [L6.3] -filling the else with a dummy statement [L2.2] [L4.3] *Assignment instead of comparison in the if statement: - (= instead of ==) $die becomes the value six instead of the comparison of $die with 6 [L2.2] [L5.2] [L4.3] *Updating variables problem - misconception about adding Boolean and numbers [L6.3]

	$count_six = $good + 1;} *Else* *$good= false;* *else {* *$good = false;* *echo "$good";*	 - false has no value (empty string) so nothing will be printed [L6.3]
print how many times six is thrown on the screen	Echo "There is $count_six times thrown six";	*Output problems: -making the output organized, with spaces and new lines [all students, all problems]
Print the thrown value on the screen	echo "The thrown value is "."$die";	*Cognitive load problem: -the thrown value is not shown on the screen [L4.3]
Increment loop counter by one	$i = $i + 1;	
print how many times in total six is thrown, on the screen	Echo "There is in total $count_six times thrown six";	*Output problems: -making the output organized, with spaces and new lines [all students, all problems]

5.3.5 Evaluation of the programming problems and which problems are specific for PHP
Most of the programming problems the students encountered when they developed the PHP programs are known problems and reported in the "known problems" section above. The "known problems" are derived from Pascal programs. The following problems are not reported in the "known problem" section. The programming problems 3 up to and including 7, which are given below, are typical problems of the programming language PHP.

1. "The empty else problem" (table 7): It is Interesting to see that none of the students had a problem with the *if-else-elseif* construct, used in the "alcohol problem", but they have a problem with just using the *if* construct (without an *else*). So they did not master this aspect of this selection construct.

2. "Assignment instead of comparison (= instead of ==) problem" for example in an *if* statement (table 7): `(if($die=6){)` $die becomes the value six instead of the comparison of $die with 6 `(if($die==6){)`. No syntax error will be generated.

3. "Output problems" to organize the output on the screen, as asked in the exercise, spaces and new lines has to be added. To add new lines in PHP the HTML tag <br> can be used in combination with the echo statement: e.g., `echo "Hello world <br>";`

4. "PHP code is on the screen problem" the PHP statements are embedded in the HTML code the PHP code part starts with a "<?PHP" or "<?" Tag and ends with a "?>" tag. One of these tags has been omitted and the PHP code is not interpreted as a PHP programming code.

5. "Blank screen problem" when testing a PHP program. A blank screen is shown, no data, no error, no title, nothing. The most common reason for this problem is that a character is missing somewhere. By simply leaving out a: " ' ","}" or a ";" somewhere, the PHP program would not work. No error or message is shown, just a blank screen.

6. "Week syntax checker problem" is related to the "blank screen problem" the Usbwebserver software has poor abilities to detect and report syntax errors. In case of a forgotten ";" the

Usbwebserver software sometimes gives the message “kan pagina niet weergeven” (can not display page) and no more detailed information of what is wrong. A more advanced syntax checker can be of help to write the right syntax. The Usbwebserver software sometimes reports a “parse error” and a line number of the line following the line with the problem [L1.1],[L7.2.2]. This was frustrating for some students [L7.1].

7. “PHP page does not stop loading problem” most often this problem is caused by infinite loops in the PHP program caused by not updating the loop counter or overwriting the counter value e.g., when starting a new (nested) loop. The PHP page gets an infinite length and can not get loaded by the internet browser program.

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) analysing 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 don not have to check their own work because the ITS will point out their errors to them. They also don 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.

Table 8. 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

Table 8 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 the programming language PHP

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 (table 8: 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 students 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 below:

6.2.1 If you want to write something on the screen then you can use the echo statement.
Example of the echo statement.
```
Echo “Hello world”;
```
On the screen appears:
```
Hello world
```
More about the echo statement: click here.

6.2.2 If you want to make the output easier to read then use spaces and newlines (<br>)
```
echo “Hello world ,<br>”;
echo “This is PHP.<br>”;
```
<br> is an HTML tag that generates a new line, on the screen appears:
```
Hello world,
This is PHP.
```
More about the echo statement: click here.

6.2.2 If you want to know the content of a variable, you can put it on the screen.
```
$name = “John”;
echo “Hello, I am $name.”;
```
on the screen appears:
```
Hello, I am John.
```

6.2.3 If you want to use a value you can use a variable
The use of values can be done by using variables in PHP. Use the following steps:

1. A variable begins with a $
2. You can give them a value by the assign statement: `$age = 16;`
3. When you become a year older you can increment the age by one: `$age = $age + 1;` The new value of `$age` is the current value of `$age` incremented with one.
4. Be aware that a variable must have a value before you can use it (e.g. `$age = 16;`)
5. more rules and examples of variables: click here

6.2.4 If something has to be done more than once then you can use a loop.
There are two possibilities to use a loop in PHP: the *while* loop (click here) and the *for* loop (click here).

6.2.5 If you want to use a while loop then take the following steps:
You can see a while loop as a repeating *if* statement (more information of the *if* statement: click here). A while loop is very flexible and can be used in many situations. An example of a while loop in PHP, for printing the table of ten on the screen, is:
```
[1] $i=0;
[2] While($i<=10){
[3]     $table_ten=$i*10;
[4]     echo “$i*10=$table_ten<br>”;
[5]     $i=$i+1;
[6] }
```
On the screen appears:
```
1*10=10
```

```
2*10=20
3*10=20
4*10=30
5*10=40
6*10=50
7*10=60
8*10=70
9*10=80
10*10=100
```

The steps you can take to make a while loop are:

1. While a condition or situation is true [2] `While ($i<=10){`
2. Take care that the variables used in the condition of the *while* have a beginning- or initial value [1] `$i=0;`
3. Do the action which has to be done [3] `$table_ten=$i*10;`
   [4] `echo "$i*10=$table_ten";`
4. Make sure that the condition of the while loop changes (otherwise the loop will never end). [5] `$i=$i+1;`
5. Put the statements assiociating to the while between `{ }` [2] `While ($i <= 10){` [6]`}`
6. More examples of *while* loops: click here.
7. Other iteration or loop constructions: click here.

6.2.6 If you want to use a for loop then take the following steps:

A for loop is very flexible and can be used in many situations. The example of the table of ten, now made with a for loop:

```
[1] for ($i=1;$i <= 10;$i++){
[2]     $table_ten = $i * 10;
[3]     echo "$i * 10 = $table_ten <br>";
[4] }
```

The steps you can take to make a for loop are:

1. `$i` becomes the starting (or initial) value 1.
2. The statements between { and } [4] will be executed: [2] `$table_ten = $i * 10;`
   [3] `echo "$i * 10 = $table_ten";`
3. Line [1] wil be executed again by incrementing $i by 1 ($i++), The condition `$i <=10` will be tested. When the condition is true the statements between `{ }` will be executed again till `$i` becomes the value `11`.
4. More examples of *for* loops: click here.
5. Other iteration or loop constructions (e.g. the `while` loop): click here.

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. Below are the questions the students asked themselves during the problem solving of the programming exercises and a possible reaction of the ITS (given in *italics*):

* 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.

6.4 Interactive session with a hypothetical ITS for PHP in which our recommendations are implemented

An interactive session with a hypothetical ITS for PHP in which our recommendations are implemented is taken from the logs of George [L4.3] "the die-problem". His steps, goals and plans and our recommended reaction of the ITS *(given in italic)* are given below:

[L4.3] Something that can count the number of six and then show it on the screen:

```
if (rand(1,6) = 6) {
Echo "6 is thrown";
}
Else {
}
```

[L4.3] How can I make it do nothing? (*FAQ: How can I let it do nothing? ITS: gives an example of an "if" without an "else" program construct)*
[L4.3] Lets make a variable which every time you throw, say….. increment with one

```
$number = 0;
$number = $number + 1;
Echo $number
```

[L4.3] (George tests the program, the program has the following code:)

```
      <?php
      $number = 0;
      if (rand(1,6) = 6){
           echo "6 is thrown ";
           }
      else {
                $ number = $number + 1;
      }
      echo $number
      ?>
```

On the screen appears: "can not display page"
(ITS: syntax error ; is missing on line 18)

(ITS: explanation of the "Assignment instead of comparison (= instead of ==) problem" in line 12)

```
if rand(1,6) == 6;
```

[L4.3] How can I let it stop on that equal sign? *(FAQ: How can I let it stop?)*
[L4.3] Now it keeps on going
[L4.3] Lets say, if it is 50 then .....

`If ($number = 50){` *(FAQ: How can I let him do something 50 times? ITS: misconception about while, confusion with if.)*

[L4.3] (George looks in his book for an example of the while construct)

```
While ($number <= 50) {
```

[L4.3] What does $i++ mean ($i++ is used by the example in the book)? *(ITS: gives explanation of $i++)*
[L4.3] Are quotes necessary when using the echo? *(ITS: gives an example of the echo statement)*
[L4.3] Making the output well organized *(FAQ: How to make the output well organized?)*

```
Echo "6 is thrown <br>";
```

[L4.3] It only says that 6 is thrown, it don't count how many times

```
$number2 = 0;
$number2 = $number2 + 1;
echo "There is $number2 times 6 thrown till now. <br>";
```

Finally the result of the program is:

```
There is 0 times thrown 6 till now.
There is 0 times thrown 6 till now.
.
.
There is 1 times thrown 6 till now.
There is 1 times thrown 6 till now.
.
.
There is 8 times thrown 6 till now.
50
There is 8 times thrown 6.
```

The final program code is:

```
<?php
$number = 1;
$number2 = 0;
while ($number <= 50){
if (rand(1,6) == 6){
      $number2 = $number2 + 1;
      $number = $number + 1;
      echo "There is $number2 times thrown till 6 now. <br>";
      }
else {
      $number = $number + 1;
      echo "There is $number2 times thrown till 6 now. <br>";
      }
}
echo "$number<br>";
echo "There is $number2 times thrown 6. <br>";
?>
```

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 “week syntax checker problem”. The “week 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.

References

Anania, J. (1983). The influence of instructional conditions on student learning and achievement. In H.J. Walberg & T.N. Postelthwaite (Eds.) *Evaluation in Education: An International Review Series 7(1)* (pp. 1-92). New York: Pergamon Press.

Anderson, J.R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Anderson, J. R. (1987). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review,* 94, 192- 210

Anderson, J. R. (1993). *Rules of the Mind.* Mahwah, NJ: Lawrence Erlbaum Associates.

Anderson, J. R. (1993). Problem solving and learning. *American Psychologist*, 48, 35-44.

Anderson, J. R., Conrad, F. G.,& Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, 13, 467–506.

Anderson, J. R., Boyle, C. F., Corbett, A., & Lewis, M. W. (1990). Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, 42, 7-49.

Anderson, J. R., Boyle, C. F., Farrell, R., & Reiser, B. J. (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed.), *Modeling Cognition*, Wiley.

Anderson, J.R., Corbett, A.T., Koedinger, K.R., & Pelletier, R. (1995). Cognitive Tutors: Lessons learned. *The Journal of the Learning Sciences 4(2)*, 167-207.

Anderson, J. R. Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, 8, 87-129.

Anderson, J. R., & Thompson, R. (1989). Use of analogy in a production system architecture. In S. Vosniadou & A. Ortony (Eds.), *Similarity and analogical reasoning.* Cambridge: Cambridge University Press.

Bloom, B.S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher, 13*, 3-16.

Booth, S. (1992). *Learning to program: A phenomenographic perspective*.(pp 43) Goteborg: Acta Universitatis Gothoburgensis.

Brown, J. S. (1985). Idea amplifiers: new kinds of electronic learning. Educational Horizons, 63, 108-112.

Butler, M., & Michael Morgan, M.(2007). Learning challenges faced by novice programming students studying high level and low feedback concepts. Faculty of Information Technology Monash University.

Carbonell, J. G. (1984). Learning by analogy: Formulating and generalizing plans from past experience. In R. S. Michalsky, J. G. Carbonell, & T. M. Mitchell (Eds.), Machine learning: *An artificial intelligence approach.* Vol. 1 (pp. 137-161). Berlin: Springer-Verlag.

Carbonell, J. G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalsky, J. G. Carbonell, & T. M. Mitchell (Eds.), Machine learning: *An artificial intelligence approach*. Vol. 2 (pp. 371-392). Los Altos, CA: Morgan Kaufman Publishers.

Collins, A., & Brown, J. S. (1987). The computer as a tool for learning through reflection. In H. Mandl & A. M. Lesgold (Eds.) *Learning Issues for Intelligent Tutoring Systems*. Springer-Verlag, New York.

Collins, A., Brown, J. S. & Newman, S. (1989). Cognitive apprenticeship: Teaching students the craft of reading, writing, and mathematics. In L. B. Resnick, (Ed.) *Knowing, learning, and instruction: Essays in honor of Robert Glaser. Hillsdale*, NJ: Erlbaum.

Collins, A., Brown, J.S. & Newman, S.E. (1990). Cognitive apprenticeship. In L.B. Resnick (Ed.). *Knowing, learning and instruction*. Hillsdale, NJ: Erlbaum.

Corbett, A.T.,Mclaughlin, A., & Scarpinatto, C. (1999). Cognitive Tutors in High School and College, Human Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA, E-mail: corbett+@cmu.edu (Received 16 November 1999; in final form 30 June 2000).

Corbett, A. T., & Trask, H. J.(2000). Instructional Interventions in computer-based tutoring: Differential impact on learning time and accuracy. In: T. Turner, G. Szwillus, M. Czerwinski and F. Paterno (eds.), CHI 2000 Conference Proceedings.

Dalbey, J., Toumiaire, E, & Linn, M. C. (1985). *Making programming instruction cognitively demanding: An intervention study* (ACCCEL report). Berkeley: University of Califomia, Lawrence Hall of Science.

Davies, S.P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies,* 39, 237–267.

Détienne, F. (1990). Expert programming knowledge: A schema based approach. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 205–222). London: Academic Press.

du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research, 2,* 57-73.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J.C. Spohrer (Eds.), (pp. 283–299). Hillsdale, NJ: Lawrence Erlbaum.

du Boulay, B., O'Shea, T., & Monk, J. (1989). The black box inside the glass box: presenting computing concepts to novices. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 431–446). Hillsdale, NJ: Lawrence Erlbaum.

Ehrlich, K., & Soloway, E. (1984). An empirical investigation of the tacit plan knowledge in programming. In J. Thomas & M. L. Schneider (Eds.), *Human factors in computer systems* (pp. 113- 133). Norwood, NJ: Ablex Publishing Corp. of Computer Assisted Learning, 3, 214-223.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol Analysis: Verbal Reports as Data* (revised edition). MIT Press, Cambridge, Mass

Gilmore, D.J. (1990). Expert programming knowledge: A strategic approach. In J.M. Hoc,T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 223–234). London: Academic Press.

Hesketh, B., Andrews, S., & Chandler, P. (1989). Opinion---Training for transferable skills: The role of examples and schema. ETTI, 26, 156-165.

Hoc, J.M. (1989). Do we really have conditional statements in our brains? In E. Soloway & J.C. Spohrer (Eds.), Studying the novice programmer (pp. 179–190). Hillsdale, NJ: Lawrence Erlbaum.

Jeffries, R., Turner, A. A., Poison, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 255-284). Hillsdale, NJ: Edbaum Associates.

Johnson, L., and Soloway, E. (1965). PROUST: Knowledge-based program understanding. *IEEE Trans. Softw. Eng. II,* 3, 267-275.

Jones, A. (1982). *Mental models of a first programming language* (CAL research group technical report, no. 29). Milton Keynes: Institute of educational Technology, England : Open University.

Kahney, H. (1989). What do novice programmers know about recursion? In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 209–228). Hillsdale, NJ: Lawrence Erlbaum.

Kassenaar, P.(2005). *PHP Keuzemodule programmeren.* Bodegraven: Instruct

Kessler, C.M., & Anderson, J.R. (1989). Learning flow of control: Recursive and iterative procedures. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 229–260). Hillsdale, NJ: Lawrence Erlbaum.

Kieras, D. E., & Bovair, S.(1986). The acquisition of procedures from text: A production system analysis of transfer of training. *Journal of Memory and Language, 25,* 507-524.

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research, 2,* 429-458.

Kurland, D.M., Pea, R.D., Clement, C., & Mawby, R. (1989). A study of the development of programming ability and thinking skills in high school students. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 83–112).

Hillsdale, NJ: Lawrence Erlbaum. Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher, 14,* 14-29.

Krammer, H. P. M., Van Merriënboer, J. J. G., & Maaswinkel, R. M. (1994). Plan-based delivery composition in Intelligent Tutoring Systems for introductory computer programming. *Computers in Human Behavior,* 10, 139-154.

Linn, M.C., & Dalbey, J. (1989). Cognitive consequences of programming instruction. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 57–81). Hillsdale, NJ: Lawrence Erlbaum.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. *Computing Surveys,* 13, 121-141.

Nassi, I., & Shneiderman, B. (1973). Flowchart techniques for structured programming, Department of Computer science, state University of New York at stony brook New York

Newell, A. (1990). United Theories of Cognition. Cambridge, MA: Harvard University Press.

Newell, A., & Simon, H. A. (1972). Human problem solving. Prentice Hall Inc., Englewood Cliffs, New Jersey.

Pea, R. D. (1986). Language-independent conceptual 'bugs' in novice programming. *Journal of Educational Computing Research, 2,* 25-36.

Pea, R. D., & Kurland, M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology, 2,* 131-168.

Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1989). Conditions of learning in novice programmers. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 261–279). Hillsdale, NJ: Lawrence Erlbaum.

Pirolli, P. L. (1985). *Problem solving by analogy and skill acquisition in lhe domain of programming.* Unpublished doctoral dissertation. Carnegie Mellon University.

Phillips, R. (2011). Code Understanding for an Intelligent Tutoring System, North Carolina State University.

Proctor, R. W., & Reeve, T. G. (1988). The acquisition of task-specific productions and modification of declarative representations in spatial-precueing tasks. *Journal of Experimental Psychology: General,* 117, 182-196.

Putmam, R. T., Sleeman, D., Baxter, J. A., & Kuspa, L. K. (1986). A summary of misconceptions of high school BASIC programmers. *Journal of Educational Computing Research, 2,* 459-471.

Reder, L. M., & Anderson, J. R. (1980). A comparison of texts and their summaries: memorial consequences. Journal of Verbal Learning and Verbal Behavior, 198, 121-134.

Reed, S. K., Dempster, A. and Ettinger, M.(1985). Usefulness of analogous solutions for solving algebra word problems. *Journal of Experimental Psychology: Learning, Memory and Cognition, 11,* 106-125.

Reder, L. M., Charney, D. H. & Morgan, K. I. (1986). The role of elaborations in learning a skill from an instructional text. *Memory and Cognition*, 14, 64-78.Rist, R. S. (1989). Schema creation in programming. *Cognitive Science,* 13, 389-414.

Robins, A., Rountree, J., Rountree, N.(2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education,* 213,137-172.

Rogalski, J., & Samurçay, R. (1990). Acquisition of programming knowledge and skills. In J.M. Hoc, T.R.G. Green, R. Samurçay & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 157–174). London: Academic Press.

Ross, B. H. (1984). Remindings and their effects in learning a cognitive skill. *Cognitive Psychology*, 16, 371-416.

Samurçay, R. (1989). The concept of variable in programming: Its meaning and use in problem solving by novice programmers. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 161–178). Hillsdale, NJ: Lawrence Erlbaum.

Sleeman, D., Putnam, R. T., Baxter, J. A., & Kuspa, L. K. (1986). Pascal and high school students: A study of errors. *Journal of Educational Computing Research, 2,* 5-24.

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

Soloway, E., Bonar, J., & Ehrlich, K. (1989). Cognitive strategies and looping constructs. In E. Soloway & J.C. Spohrer (Eds.). *Studying the novice programmer* (pp. 191–207). Hillsdale, NJ: Lawrence Erlbaum.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10, 595–609.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1983). What do novices know about programming? In B. Shneiderman & A. Badre (Eds.), *Directions in human-computer interactions* (pp. 27–54). Norwood, NJ: Ablex.

Spohrer, J.C., Soloway. E. & Poppe, E. A. (1985). Goal/plan analysis of buggy Pascal programs. *Hum.-Compuf. Interaction I,* 2, 163-207.

Spohrer, J.C., & Soloway, E. (1989). Novice mistakes: Are the folk wisdoms correct? In E. Soloway & J.C. Spohrer (Eds.). *Studying the novice programmer* (pp. 401–416). Hillsdale, NJ: Lawrence Erlbaum.

Spohrer, J.C., Soloway, E., & Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 355–399). Hillsdale, NJ: Lawrence Erlbaum.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning, *Cognitive Science*, 12, 257-286.

VanLehn, K. (2006). The behaviour of tutoring systems. *International Journal of Artificial Intelligence in Education*, vol. 16, Issue 3, pp. 227 -265.

Van Merriënboer, J. G., & Paas, G. W. C. (1990). Automation and Schema Acquisition in Learning Elementary Computer Programming: Implications for the Design of Practice. University of Twente, *Computers in Human Behavior, Vol. 6,* pp. 273-289, 1990.)

Van Someren, M. W., Bernard, Y. F., & Sandberg, J. A. C. (1994). *The think aloud method: A practical guide to modeling cognitive processes.* London: Academic Press.

Vessey, I. (1987).On Matching Programmers' Chunks with Program Structure: An Empirical Investigation. *International Journal of Man-Machine Studies*, vol. 27, (pp 65-89)

Appendix 1

[L1.1] session [1] 090312	docent/anderen	scherm	programma	commentaar
(test de apparatuur en maakt de instellingen voor camstudio)				

2.50 Leest de opdracht
 	
Opdracht1
Maak een programma met HTML en PHP *dat (met een while loop) bepaalt wat de uitkomst is als je de getallen 1,2,3….tot en met een getal dat je van te voren in de code vastlegt optelt, en daarna de uitkomst op het scherm toont. (Als bijvoorbeeld het getal de waarde 5 heeft, wordt er uitgerekend hoeveel 1+2+3+4+5 is en de uitkomst 15 wordt op het scherm afgedrukt.) Laat ook zien hoe de berekening tot stand komt.Zie het voorbeeld hieronder:*
 	
`Het totaal van 1 tot en met 5 is:`
`1+2+3+4+5 = 15`				

(maakt praatje met buurman)				
Ik snap er niks van, je moet toch hardop denken. Ik snap er helemaal niets van.				
5.20				
Mevrouw, waar moet ik het opslaan?		<html> <head> <title> opdracht 1 </title> </html>		
	In de root			
		</head> <body>		
		</body>		

(maakt een ritme met geluidjes)				
[Waar staat ie, waar staat het dingetje?*(zoekt in het boek)*				Zoekt een voorbeeld van de while in het boek op.
10.00	Heb je een idee hoe je het aan gaat pakken?			
Niet echt				
12.00	Ben je het aan het doorlezen?			
hmmm				
		<?php		
		?>		
16.18		$i = 1;		While loop: loop teller initiële waarde geven.
		while ($i <= $getal)		While statement met teller en stop voorwaarde met variabele
		$getal = 6;		Nieuwe variabele een initiële waarde geven
		Aanpassen: while ($i <= $getal) while ($i <= $getal){		Syntax van de while controleren
		echo "$i+1";		Loopteller" + 1" op het scherm weergeven
i krijgt de waarde van dollar I plus 1				
		$i = $i + 1; }		Loopteller 1 ophogen While afsluiten met }

(slaat usbwebserver op en localhost, slaat sourcecode op als php file)		1+12+13+14+15+16+1	<html>
			<head>
			<title> opdracht 1 </title>
			</head>
			<body>
			<?php
			$i = 1;
			$getal = 6;
			while ($i <= $getal){
			echo "$i+1";
			$i = $i + 1;
			}
			
			?>
			</body>
			</html>	

21.28		*Aanpassen:* echo "$i+1"; echo "$i+"		Uitvoer naar het scherm aanpassen nav test resultaat
testen		1+2+3+4+5+6+		testen
		Aanpassen: while ($i <=getal){ while ($i <$getal){		Bovengrens van de while aanpassen nav testresultaat

		Echo $getal		

testen		1+2+3+4+5+6	<?php
			$i = 1;
			$getal = 6;
			$p = 0;
			while ($i < $getal){
			echo "$i+";
			$i = $i + 1;
			
			}
			?>	

		echo "$getal = ";		
		echo "Het totaal van 1 tot en met $getal is: "		

testen		Parse error: syntax error, Unexpected T_WHILE, expecting '.' or ';'in D:\.....\opdracht1.php.php on line 10	<?php
			$i = 1;
			$getal = 6;
			*[9]*echo "Het totaal van 1 tot en met $getal is:<br>"
			*[10]*while ($i < $getal){
			echo "$i+";
			$i = $i + 1;
			
			}
			echo "$getal = ";
			
			?>
				Syntax fout ; bij echo statement vergeten

		echo "Het totaal van 1 tot en met $getal is:";		Uitvoer aanpassen aan vraag van de opdracht

testen		Het totaal van 1 tot en met 6 is: 1+2+3+4+5+6=	<?php
			$i = 1;
			$getal = 6;
			echo "Het totaal van 1 tot en met $getal is: ";
			while ($i < $getal){
			echo "$i+";
			$i = $i + 1;
			
			}
			echo "$getal = ";
			
			?>	

		echo "Het totaal van 1 tot en met $getal is:
";		
testen		Het totaal van 1 tot en met 6 is: 1+2+3+4+5+6=	<?php $i = 1; $getal = 6;	

			echo "Het totaal van 1 tot en met $getal is:
			 ";
			*[10]*while ($i < $getal){
			echo "$i+";
			$i = $i + 1;
			
			}
			echo "$getal = ";
			
			?>	

26.00 Dave, ik heb al dit, ik heb alleen de uitkomst niet.				Bekijkt het resultaat, mist het resultaat van de optelling
	Heb je hem al? Ik zie nog geen uitkomst			
Nee, dat klopt, dat weet ik niet hoe je dat moet doen, eigenlijk.				
Hee, hoe doe je de uitkomst?				
		$p = 0		
		$p = 0 + $i;		Voegt variabele toe voor het optellen van de gevraagde waarden

		Aanpassen: echo "$getal = ";
		 echo "$getal = $p";		

testen		Parse error: syntax error, Unexpected T_WHILE, expecting '.' or ';'in D:\.....\opdracht1.php.php on line 10	<?php
			$i = 1;
			$getal = 6;
			$p = 0
			*[10]*echo "Het totaal van 1 tot en met $getal is:
			";
			while ($i < $getal){
			echo "$i+";
			$p = 0 + $i;
			$i = $i + 1;
			
			}
			$p = $p + $getal;
			echo "$getal = $p ";
			
			?>	

Bekijkt regel 10				
		Break;		
testen		Parse error: syntax error, Unexpected T_WHILE, expecting '.' or ';'in D:\.....\opdracht1.php.php on line 10	<?php $i = 1; $getal = 6;	

			$p = 0
			*[10]*echo "Het totaal van 1 tot en met $getal is:
			";
			while ($i < $getal){
			echo "$i+";
			$p = 0 + $i;
			$i = $i + 1;
			break;
			}
			$p = $p + $getal;
			echo "$getal = $p ";
			
			?>	

		Verwijdert: break;		
Bekijkt de code				
	Hoe gaat het?			
Nou, hij doet het niet.				
	Want, regel 10?			
Net was regel 10 wel gewoon goed.				
Ik heb alleen dit *(regel 9) en $p = 0 +$i;* toegevoegd. En nou zegt ie dat regel 10 fout is. Regel 10 was net nog goed.				
	Wat wil je daar dan ? *wijst op regel 9)*			
O, ja maar dat is geen regel 10.		*Aanpassen: $p = 0* *$p = 0;*		Syntax fouten kunnen ook in een ander regel zitten dan de regel die aangegeven wordt.

testen		Het totaal van 1 tot en met 6 is:
		1+2+3+4+5+6=5
			$i = 1;
			$getal = 6;
			$p = 0;
			*[10]*echo "Het totaal van 1 tot en met $getal is:
			";
			while ($i < $getal){
			echo "$i+";
			$p = 0 + $i;
			$i = $i + 1;
			}
			$p = $p + $getal;
			echo "$getal = $p ";
				Moeite met variabel $p aanpassen mbv variabele zelf ($p=$p+$i)

		Aanpassen: $p = 0 + $i; $p = $p + $i;		Ontdekt de fout nav de uitvoer op het scherm.
testen 31.27		Het totaal van 1 tot en met 6 is: 1+2+3+4+5+6=15	$i = 1; $getal = 6; $p = 0;	Controleert het resultaat, en ziet dat de laatste waarde niet opgeteld wordt.

			echo "Het totaal van 1 tot en met $getal is:
			";
			while ($i < $getal){
			echo "$i+";
			$p = $p + $i;
			$i = $i + 1;
			}
			
			echo "$getal = $p ";	

		$p = $p + $getal;		

testen		Het totaal van 1 tot en met 6 is:
		1+2+3+4+5+6=21
			$i = 1;
			$getal = 6;
			$p = 0;
			echo "Het totaal van 1 tot en met $getal is:
			";
			while ($i < $getal){
			echo "$i+";
			$p = $p + $i;
			$i = $i + 1;
			}
			$p = $p + $getal;
			echo "$getal = $p ";
				Lost dit op door buiten de loop deze waarde bij het resultaat op te tellen. Uitvoer resultaat bekijken en interpreteren en dan goede oplossing toevoegen.

Ik heb hem gewoon. Ik heb hem mevrouw.				
	Ja, ok. Dan heb je hem opgeslagen en stop je de recording en sluit niet af, ik sla het voor je op.			