University of Antwerp (2009)
URL: http://www.eduvizor.org/research/pdf/moons2009-1.pdf 51 pages

Applying learning theory principles in the design of program visualization software

Jan Moons[1]

Department of Management Information Systems, Faculty of Applied Economics
Universiteit Antwerpen
Antwerpen, Belgium

Carlos De Backer[2]

Department of Management Information Systems, Faculty of Applied Economics
Universiteit Antwerpen
Antwerpen, Belgium

Abstract

Teaching introductory programming presents a serious challenge to CS1 teachers the world over. Several reports note the unusually high failure rate of introductory programming courses. This article has three goals. First, it aims to provide an overview of the issues students face in a CS1 course. Next, it will present an overview of learning theory and show how learning theory guidelines can be applied in a program visualization component that will improve students' understanding of object-oriented programming concepts. Finally, the article will present a new program visualization plug-in that adheres to the principles found in learning theory, and compare this tool with other program visualization tools.

Keywords: Program visualization, CS1, Java programming

1 Introduction

The difficulty of programming has been a mainstay of computer science education research for decades. In Degrace & Stahl (1998), programming is called a "wicked problem" that is *often too big, too ill-defined, and too complex for easy comprehension and solution* (Petre & de Quincey, 2006). This level of complexity is widely recognized in literature (Jeffries *et al.*, 1981; Kim & Lerch, 1997) - or, as stated by Kurland *et al.* (1986), *programming seems to demand complex cognitive skills such as procedural and conditional reasoning, planning, and analogical reasoning.* The complexity of programming is also apparent in the reactions of beginning programmers to this particular computer science area. As stated by Baldwin & Kuljis (2000), the majority of students find computer programming a difficult and complex

[1] Email: jan.moons@ua.ac.be
[2] Email: carlos.debacker@ua.ac.be

cognitive task. This is not a recent phenomenon - the difficulties of learning computer programming have been acknowledged in literature for more than a quarter century (Mayer, 1981).

In recent years, two reports stand out that elicit the scope of the problem. The first report is the result of a broad international study under the lead of Michael McCracken entitled "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students" (McCracken *et al.*, 2001). In this report, the authors test the programming competency of students after they have completed their first one or two programming courses. More specifically, the students had to program three exercises. The first exercise concerned the evaluation of a postfix expression, the second concerned the evaluation of an infix expression from left to right and the third exercise concerned an infix expression with parenthesis precedence. The students received training in the operation of postfix and infix expression before the evaluation. In total, 217 students from four universities took the test.

The solutions provided by the students were scored on several criteria, which were aggregated in a *general evaluation score*. Disappointingly, the average general evaluation score for all students, for all exercises, across all schools was 22.9 out of 110 (standard deviation 25.2).

A second international study was performed under the auspices of Raymond Lister, entitled "A multi-national study of reading and tracing skills in novice programmers" (Lister *et al.*, 2004). This study was created to check the hypothesis that beginning students lack the important routine programming skill of tracing (or "desk checking") through code. Over 600 students from seven countries were tested on twelve multiple choice questions of two types. The first type required the students to predict the outcome of a piece of code. The second type required students to choose the correct completion of a piece of code from a small set of possibilities. Although the results were not as bad as the McCracken study, the results were still unexpectedly poor. As an indication, more than a quarter of the students failed the test ...

The reports by McCracken *et al.* (2001) and Lister *et al.* (2004) show that both program comprehension and program generation are problematic. These reports are confirmations of earlier studies. For instance, the problematic tracing skills had already been described by Perkins *et al.* (1989). More information on the difficulty of a first programming course can be found in Milne & Rowe (2002); Hristova *et al.* (2003); Thompson (2004); Ahmadzadeh *et al.* (2005); Garner *et al.* (2005) and Jadud (2006).

Based on the widespread nature of these problems, we have set ourself the goal of solving the following research question:

> *Can we design and develop an integrated software component based on learning theory that can help in teaching the fundamental concepts of object-oriented programming in Java to novice programmers?*

This research question follows from several observations:

(i) The difficulty of learning to program is very high as indicated above.

(ii) Learning theory can provide essential theoretical insight into how students

learn and hence how educational material should be designed.

(iii) The leading paradigm today is object orientation, which brings along its own set of difficulties which will be described further along in the paper.

(iv) The language that currently enjoys the widest range of adoption is Java. [3]

(v) The development of a tool rather than a methodology or way of teaching is a choice of the authors.

In section 2 we present a theoretical backdrop for the remainder of the article. More specifically, we look at the leading paradigms of learning and how they apply to computer science education. Next, in section 3, we evaluate different CS1 tool categories based on the high level criteria that are deduced from the learning paradigms, i.e. the need for the presentation of schema of language constructs and the need for a high level of interaction. In section 4, we translate the lessons provided by cognitive load theory and perception theory into four design criteria. We also show how we have applied these criteria in the design of our own tool. Section 5 will present schema that can be used to visualize various problematic Java language features. In section 6 various available program visualization tools are compared based on the language features they are able to present and based on the design criteria we have defined earlier. Finally, in section 7 we present our conclusive remarks.

2 Theories of learning and computer science education

As CS1 teachers we know from first-hand experience how difficult the first steps toward computer science excellence can be. Although anecdotally we have a notion of the specific problems our students face during the CS1 course, we know little about the cause of these problems and how they could be mitigated. In other words, we are in need of a more thorough evaluation of the problem domain. In this article, which is about the design and development of a CS1 tool, we want to expand our horizon beyond the field of computer science and take a look at theory that discusses how students learn, and by extension, how instructional materials should be designed. This chapter will highlight theories of learning, educational psychology and instructional design that might help in understanding the problems mentioned in chapter 1. These are all major research area's with very rich and vibrant research communities and it is not our goal to describe all theories in intricate detail or to confirm or reject these theories. Rather, we will provide an - understandably - much simplified overview of the most important theories in so far as they can help us in the rest of this research project.

For each learning paradigm, we will also assess their respective impact on the field of computer science learning and education. A first observation is that relatively few research reports directly reference these theories. As mentioned by East & Wallingford (1997), *there is indeed little discussion of the teaching of programming that relates to pedagogy and almost none that addresses how the process of learning might or should affect instruction.* This strikes us as odd - why would

[3] Check http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

computer science education be that different from other types of education? One of the reasons for this observation is probably that most researchers involved in CSE research stem from a background in computer science, rather than in educational psychology. Whatever the cause, there is certainly value to be had by incorporating these theories in computer science education.

There are currently two main paradigms of learning - cognitivism (along with its predecessor behaviorism) and constructivism. Many theories based on these paradigms exist, but for our discussion we will limit ourselves to these paradigms. Section 2.1 will discuss the first paradigm - cognitivism, while section 2.2 will discuss the second paradigm, constructivism.

2.1 Cognitivism

2.1.1 Cognitivism as a learning theory - background

Cognitivism as a research area investigates the way our mind works[4]. It is concerned with the inner workings of human cognitive architecture. Most cognitive scientist now converge on the basic model of cognitive architecture as proposed by Atkinson & Shiffrin (1968). The original model is reproduced in figure 1. **Atkinson** divides memory into three stores. The sensory register, the short-term store and the long-term store. Information from the environment enters the mind through the sensory register. The **sensory register** has an extremely limited decay period, i.e. the time it takes for registered data to disappear from the register. Atkinson mentions a decay period for visual information of several hundred milliseconds. The short-term store is also referred to as "**working memory**". All conscious processing happens in working memory. When referring to cognitive load, researchers refer to the load on working memory. Working memory also has a limited decay period. Atkinson mentions a decay for items registered in the audio-verbal-linguistic store of 15 to 30 seconds. The final component is **long-term memory**, which serves as a virtually inexhaustible and permanent store of information.

Most research on human cognitive architecture is focused on working memory, long-term memory and the interaction between them. Working memory is limited in two ways. As we already mentioned, information in working memory is only retained for a few seconds. Working memory is also constrained in capacity. In his famous article, George Miller calculated the number of items that can be held simultaneously in working memory to be **seven, plus or minus two** depending on circumstances (Miller, 1956). Other evidence suggests an even more limited capacity of four plus or minus one (Cowan, 2001).

For cognitivists, learning is defined as *as a change in long-term memory* (Kirschner *et al.*, 2006). To be more precise, learning is the creation of **schemas** in long-term memory. Schemas are structures in long-term memory that categorize elements of information according to the manner in which they will be used (Sweller, 1998). In other words, a schema is an abstraction that can incorporate multiple pieces of information. Sweller provides several examples:

[. . .] Chess grand masters have schemas that categorize board pieces into patterns

[4] This is in contrast with its predecessor **objectivism**, which deals with stimulus-response mechanisms without considering the internal mechanism that deals with the response

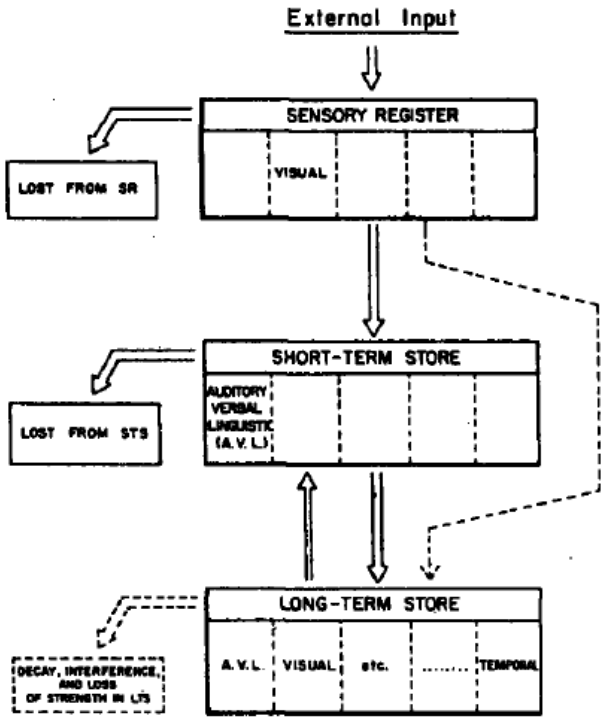

Fig. 1. The original model of cognitive architecture as presented by Atkinson & Shiffrin (1968).

that tell them which moves are appropriate. Schemas can tell us that certain objects are trees to which we can react in a common way even though no two trees have identical elements. When reading, we can derive meaning from an infinite variety of marks on a page because we have schemas that allow us to appropriately categorize letters, words, and combinations of words. Schemas provide the elements of knowledge. According to schema theory, it is through the building of increasing numbers of ever more complex schemas by combining elements consisting of lower level schemas into higher level schemas that skilled performance develops.

Besides their function of knowledge categorization, schemas also reduce **working memory load**. According to schema theory, schemas can incorporate huge amounts of information. There is no perceivable limit on the complexity of schemas. In addition, a schema is only treated as a single unit in working memory. This means that people who have created a schema for a difficult subject still have resources left in working memory when dealing with this subject, while people who have not created a schema may have already exhausted theirs.

Cognitive load theory (CLT) is a theory based on our knowledge of human cognitive architecture and implicates many practical suggestions for reducing cognitive load on the learner through good instructional design. CLT is defined by Clark *et al.* (2006) as *a universal set of principles that are proven to result in efficient instructional environments as a consequence of leveraging human cognitive learning process.*

CLT distinguishes between three types of cognitive (or working-memory) load

(Paas *et al.*, 2003). The first type is **intrinsic** cognitive load - load that is inherent to the nature of the subject. There is nothing fundamental that instructional design can do to reduce this type of load, but instructors can *artificially* reduce inherent cognitive load by scaffolding difficult subjects, i.e. by gradually exposing the student to more complex views on the subjects (van Merrinboer *et al.*, 2003).

The second type of load is called **extraneous** cognitive load or ineffective load. This type of load is caused by poorly designed instruction. One of the main goals of CLT is to formulate directives that can reduce extraneous cognitive load. The third type of load is called **germane** or effective load, which is caused by processes that contribute to the construction and automation of schemas (Paas *et al.*, 2003). In summary, CLT seeks to artificially reduce intrinsic cognitive load through segmenting and scaffolding, reduce extraneous cognitive load through solid instructional design and use available working memory capacity by applying germane load to enhance schema creation.

2.1.2 Cognitivism in computer science education

For novice programmers, intrinsic cognitive load is undeniably high. For example, the first thing most educators in a CS1 course show their students is a "Hello World" example in their favorite programming language. Let's take Java as an example. The problem is illustrated with a sense of humor in Bailie *et al.* (2003): *from the first line of a Java program you know we are in serious trouble: "public static void main(String[ ] args)." We have visibility modifiers, return types, method names, a class, parameters and arrays and we haven't started the program.* Many similar languages face the same problem - even for writing extremely simple examples, students need to master a large amount of new concepts and techniques.

The problem is not just that there are a whole host of concepts to be learned, there is also a **high level of element interactivity**, which means that the elements depend on each other. For instance, in order to implement a method, one must also know about parameters, return values, access modifiers etc. This means that many elements must be present in working memory at once, increasing intrinsic cognitive load. In fact, many constructs that are considered to be the building blocks of object oriented programming are abstract, high-level concepts (such as a "method" or a "class") covering many lower-level concepts.

Keeping in mind CLT, students that need to master higher-level concepts without overloading working memory first need to master the constituent lower-level concepts. This means teachers of CS1 courses must explicitly present the schema of the constituent concepts. In other words, teachers should employ careful sequencing and scaffolding techniques when introducing concepts, starting with lower-level concepts and working up to higher-level concepts.

A **first** computer science education research topic that relates to cognitivism and schema theory is the search for characteristics that differ between novice and expert programmers. Table 1 summarizes the findings of the comparison of novice programmers' characteristics, expert programmers' characteristics and independent characteristics from the overview article by Winslow (1996).

Winslow mentions many of the concepts introduced in the previous section on CLT. For instance, the reliance of experts on deep knowledge of many mental mod-

Novice programmers	Expert programmers	Independent characteristics
• lack an adequate mental model of the area (Kessler & Anderson, 1987). • are limited to a surface knowledge of subject. • have fragile knowledge (something the student knows but fails to use when necessary) and neglect strategies (Perkins & Martin, 1986). • use general problem solving strategies (i.e., copy a similar solution or work backwards from the goal to determine the solution) rather than strategies dependent on the particular problem. • tend to approach programming through control structures • use a line-by-line, bottom up approach to problem solution (Anderson, 1985).	• have many mental models and choose and mix them in an opportunistic way (Visser & Hoc, 1990). • have a deep knowledge of their subject which is hierarchical and many layered with explicit maps between layers. • apply everything they know. • when given a task in a familiar area, work forward from the givens and develop subgoals in a hierarchical manner, but given an unfamiliar problem, fall back on general problem solving techniques. • have a better way of recognizing problems that require a similar solution (Chi *et al.*, 1981; Davies, 1990). • tend to approach a program through its data structures or objects(Petre & Winder, 1988). • use algorithms rather than a specific syntax (they abstract from a particular language to the general concept). • are faster and more accurate (Weidenbeck, 1986; Allwood, 1986). • have better syntactical and semantical knowledge and better tactical and strategic skills (Bateson *et al.*, 1987).	• Given a new, unfamiliar language, the syntax is not the problem, learning how to use and combine the statements to achieve the desired effect is difficult. • Learning the concepts and techniques of a new language requires writing programs in that language. Studying the syntax and semantics is not sufficient to understand and properly apply the new language. • Problem solution by analogy is common at all levels; choosing the proper analogy may be difficult. • At all levels, people progress to the next level by solving problems. The old saw that practice makes perfect has solid psychological basis.

Table 1
A list of novice, expert and level-independent programmer characteristics by Winslow (1996)

els, the reliance on abstracted knowledge and the ability to recognize problems that require a similar solution. Most differences between beginner and expert programmers are very much in line with cognitive load theory. As an example, in Shneiderman (1980), the authors describe a study that compares the recall of presented lines of code between novices and experts. When the presented program was real, the experts were able to recall many more codelines than the novices, but when the presented program consisted of random lines of code, there was no difference at all. This indicates that experts are able to form schema of meaningful blocks of code, while novices have to remember line by line.

A **second** computer science education topic related to cognitivism and schema theory is that of visualization. In fact, some guidelines of cognitive load theory explicitly mention the power of visualization. These are the first three (of a series of 39) guidelines presented in Clark *et al.* (2006):

(i) Use diagrams to optimize performance on tasks requiring spatial manipulations.

(ii) Use diagrams to promote learning of rules involving spatial relations.

(iii) Use diagrams to help learners build deeper understanding.

These guidelines are based on empirical research. For instance, in Marcus *et al.* (1996), the authors present a study that compares two groups of children requiring to create connections between resistors in an electrical configuration. One group is provided with textual aids, the other with graphical aids. The results were that the

diagram group completed the taks faster and with fewer errors. The authors relate the findings to CLT:

> *If the number of elements that must be processed exceeds working-memory capacity, then some elements must be combined into schemas before the material can be understood. A diagram may reduce cognitive load by providing such a schema. In a series of experiments, 3 different electrical resistor problems were given to students to complete, with instructions presented using diagrams or text. Results suggested that understanding depends on the degree of interaction among elements of information. However, if interacting elements can be incorporated into a diagrammatic schema, cognitive load will be reduced and understanding enhanced.*

In order to understand why exactly graphics can help in reducing cognitive load, we have to turn to the theory of perception. Clark *et al.* (2006) provides us with an explanation for the power of diagrams:

> *The authors state that all elements in a visual can be viewed simultaneously, unlike sentences, which must be processed sequentially one at a time. This leads to a lower visual search for tasks that involve coordination of multiple spatial elements. Greater psychological processing efficiency is the result. Likewise, diagrams provide more explicit representation of spatial tasks. A diagram requires fewer inferences because it shows spatial relationships that must be inferred from textual descriptions.*

This quote is similar in nature to the findings of Larkin & Simon (1987), who compare the properties of two different types of external representation, which they call sentential and diagrammatic. Sentential information is ordered sequentially, like the propositions in a text. Diagrammatic representations are indexed by location in a plane. According to the authors, sentential representations require a lot more cognitive effort from users, because there is a continual need to hold on to values while searching for relational information, while in diagrammatic representations information is organized by location, and much of the relational information needed to make an inference is present and explicit at a single location. But diagrams are not only beneficial for spatial *tasks*. Any concept that comprises relationships is best represented through diagrams. Again, this claim has been verified empirically Carlson *et al.* (2003).

So, in general, diagrams are a good way of teaching concepts that incorporate relations. This is interesting for CS1 educators, for what else is a running object oriented program than a collection of objects, variables, method calls and static structures that are related to one another in often complex ways.

The power of visualization is widely accepted within computer science education circles. For instance, in a survey presented in Naps *et al.* (2002), 93% of the participants of the ITiCSE2002 conference acknowledged the benefits of visualization. Further in the article, the authors present the benefits of visualization as experienced by educators (see table 2)

Visualization in CSE has been tested empirically on many occasions. For example, Smith & Webb (2000) mention that their experiment provides empirical evidence that visualization tools help improving mental models. In Hendrix *et al.* (2000), the authors present the findings of an experiment on the impact of control

Percentage	Description
90%	the teaching experience is more enjoyable
86%	improved level of student participation
83%	anecdotal evidence that the class was more fun for students
76%	anecdotal evidence of improved student motivation
76%	visualization provides a powerful basis for discussing conceptual foundations of algorithms
76%	visualization allows meaningful use of the available technology
72%	anecdotal evidence of improved student learning
62%	(mis)understandings become apparent when using visualization
52%	objective evidence of improved student learning
48%	interaction with colleagues as a benefit

Table 2
The advantages of using visualization technology according to Naps *et al.* (2002).

structure diagrams that show that the diagrams are *highly significant in enhancing the subjects' performance in [the] program comprehension task.* Other corroborative conclusions are provided in Ma *et al.* (2007).

2.2 Constructivism

2.2.1 Constructivism as a learning theory - background

This section deals with the learning paradigm of constructivism, which is based on the philosophical epistemology of *relativism.* From a relativist point of view, truth is always relative to the subject, or in our case, to the learner. Constructivism therefore advocates that knowledge is *constructed* by the learner rather than transferred by the teacher. Some of the primary authors for this paradigm are Piaget (1970), Vygotsky (1978), Bruner (1979) and von Glasersfeld (1987). To understand the roots of constructivism, we cite Piaget (1970):

> *I find myself opposed to the view of knowledge as a copy, a passive copy, of reality. In point of fact, this notion is based on a vicious circle: in order to make a copy we have to know the model that we are copying, but according to this theory of knowledge the only way to know the model is by copying it, until we are caught in a circle, unable ever to know whether our copy of the model is like the model or not. To my way of thinking, knowing an object does not mean copying it - it means acting upon it.*

Constructivism is described in laymen's terms by Ma *et al.* (2007), who state that *constructivism argues that students actively construct knowledge by combining the experiential world with existing cognitive structures.* The difference between behaviorism and constructivism has been described by Bichelmeyer and Hsu as follows (in the context of this quote, the reader can substitute "behaviorism" with "cognitivism" as both are based on the same epistemology):

> *Where behaviorism views knowledge as resulting from a finding process, constructivism views knowledge as the natural consequence of a constructive process. Where behaviorism views learning as an active process of acquiring knowledge, constructivism views learning as an active process of constructing knowledge. Finally, where behaviorism views instruction as the process of providing knowl-*

edge, constructivism views instruction as the process of supporting construction of knowledge. (Bichelmeyer & Hsu, 1999)

Instruction based on constructivist principles is often referred to as minimally-guided instruction. Specific examples of constructivist-based design strategies are discovery learning (Bruner, 1979), problem-based learning (Barrows, 1996; Savery, 2006), inquiry learning (Papert, 1980) and experiential learning (Boud, 1985). These theories are all based on the same constructivist idea that *the most personal of what [a man] knows is that which he has discovered for himself* (Bruner, 1979).

Sjoberg (2007) distinguished three types of constructivism in literature, based on what exactly is constructed.

(i) Is it our individual knowledge about the world? ("Children construct their own knowledge.")

(ii) Is it the shared and accepted scientific knowledge about the world as it exists in established science? ("Scientific knowledge is socially constructed")

(iii) Or is it the world itself? ("The world is socially constructed")

According to Sjoberg, only the first question is a problem of psychology and educational or learning theory, while the latter two are part of philosophy and epistemology. Considering only the first - and least extreme - interpretation of constructivism, the author presents the fundamental ideas of constructivism (based on Taber (2006)) as follows:

(i) Knowledge is actively constructed by the learner, not passively received from the outside. Learning is something done by the learner, not something that is imposed on the learner.

(ii) Learners come to the learning situation (in science etc.) with existing ideas about many phenomena. Some of these ideas are ad hoc and unstable; others are more deeply rooted and well developed.

(iii) Learners have their own individual ideas about the world, but there are also many similarities and common patterns in their ideas. Some of these ideas are socially and culturally accepted and shared, and they are often part of the language, supported by metaphors etc. They also often function well as tools to understand many phenomena.

(iv) These ideas are often at odds with accepted scientific ideas, and some of them may be persistent and hard to change.

(v) Knowledge is represented in the brain as conceptual structures, and it is possible to model and describe these in some detail.

(vi) Teaching has to take the learner's existing ideas seriously if they want to change or challenge these.

(vii) Although knowledge in one sense is personal and individual, the learners construct their knowledge through their interaction with the physical world, collaboratively in social settings and in a cultural and linguistic environment. (The relative stress on such factors account for the different 'versions' of constructivism [...])

This list implies that constructivism in the least extreme form should not be considered completely orthogonal to cognitivism. For instance, the conceptual structures of constructivism are akin to the schema of cognitivism (item 5). In addition, the idea of the active construction of knowledge proposed by constructivism is also present in schema theory, one of the building blocks of cognitivism. For instance, consider this quote by Sweller (1998):

> *Often, this acquisition of schemas is an active, constructive process. Reading provides a clear example. In early school years, children construct schemas for letters that allow them to classify an infinite variety of shapes (as occurs in hand writing) into a very limited number of categories. These schemas provide the elements for higher order schemas when they are combined into words that in turn can be combined into phrases, and so forth.*

One particular branch of constructivism, *constructionism*, is interesting because the primary author, Seymour Papert, is a well respected mathematician and computer scientist, active in the MIT Artificial Intelligence and Media Labs. In his book, *MINDSTORMS - Children, Computers, and Powerful Ideas*(Papert, 1980) the author presents his ideas in detail. Papert was intrigued by the affective component of constructionism. It is natural for humans to enjoy creating artifacts, and most of his work is based on this observation. The book is concerned with ways in which computers can be actively involved in the education of children, for instance through using simple programming languages such as Logo. In Harel & Papert (1991), the authors present constructionism as follows:

> *Constructionism - the N word as opposed to the V word shares contructivism's view of learning as "building knowledge structures" through progressive internalization of actions... It then adds the idea that this happens especially felicitously in a context where the learner is consciously engaged in constructing a public entity, whether its a sand castle on the beach or a theory of the universe.*

Although constructionism and constructivism are sometimes seen as two independent paradigms, in reality there is no real divide between the work of Papert (constructionist) and Piaget (constructivist), only their focus is different, with constructionism particularly paying attention to the affective component of creating artifacts. Papert, who has worked extensively with Piaget in his career, recounts the following after describing his childhood love for cars and gear systems and how they helped him conquer mathematics:

> *Many years later when I read Piaget this incident served me as a model for his notion of assimilation, except I was immediately struck by the fact that his discussion does not do full justice to his own idea. He talks almost entirely about cognitive aspects of assimilation. But there is also an affective component. Assimilating equations to gears certainly is a powerful way to bring old knowledge to bear on a new object. But it does more as well. I am sure that such assimilations helped to endow mathematics, for me, with a positive affective tone that can be traced back to my infantile experiences with cars. I believe Piaget really agrees. As I came to know him personally I understood that his neglect of the affective comes more from a modest sense that little is known about it than from*

an arrogant sense of its irrelevance. (Papert, 1980)

The differences and similarities between constructivism and constructionism are described in Ackermann (2001). With respect to Piaget, Ackermann states that *while capturing what is common in children's thinking at different developmental stages - and describing how this commonality evolves over time - Piaget's theory tends to overlook the role of context, uses, and media, as well as the importance of individual preferences or styles, in human learning and development.* With respect to Papert, she writes that *because of its greater focus on learning through making rather than overall cognitive potentials, Papert's approach helps us understand how ideas get formed and transformed when expressed through different media, when actualized in particular contexts, when worked out by individual minds.*

In conclusion, constructionism is more concerned with the specific modalities (as in, through the creation of an artifact) of knowledge creation. In addition, Papert's work is often concerned with the use of technology and computers as facilitators for artifact creation.

2.2.2 Constructivism in computer science education

One of the primary authors on constructivism in recent CSE literature is Ben-Ari (see e.g. Ben-Ari (1998, 2001, 2004)). Most of the points that Ben-Ari presents in his papers pertain to the least extreme view of constructivism, i.e. that learners construct their own models. In his work, Ben-Ari is often critical of the type of unguided learning advocated by extreme constructivists. For instance, in Ben-Ari (2001) the author presents three conclusions:

(i) Models must be explicitly taught.

(ii) Models must be taught before abstractions.

(iii) The seductive reality of the computer must not be allowed to supplant construction of models. (with this Ben-Ari implies that we should not encourage the *bricolage* type of development that characterizes the style of many novice programmers too much. Trial-and-error programming can be helpful for novices, but only when it supplements rather than supplants planning and formal methods)

The conclusions make it really obvious that Ben-Ari's take on constructivism, which is the least extreme view of constructivism, is in fact not at all orthogonal to cognitivism. Ben-Ari warns against the formation of unviable mental models by novice programmers if these models are not explicitly taught. This constrasts with the traditional constructivist approach of teaching using unguided learning techniques.

Nonetheless, while the pure form of constructivism might not be compatible with computer science education, an important idea of constructivism - the preference of active learning styles over passive learning styles - has found its way into computer science education and related areas such as math, science and engineering. For instance, the constructivist learning style of engineering students is described in Felder & Silverman (1988). The authors conclude that *learning styles of most engineering students and teaching styles of most engineering professors are incompatible*

in several dimensions. Many or most engineering students are visual, sensing, inductive, and active. [...] Most engineering education is auditory, abstract, deductive, passive, and sequential. These mismatches lead to poor student performance, professorial frustration, and a loss to society of many potentially excellent engineers. In research supportive of Felder's conclusion, Palmer *et al.* (2005) mentions that a large majority of respondents to their questionnaire (all grade A respondents and nine out of fourteen grade D respondents) preferred a kinaesthetic (active) learning style to a passive audio-visual learning style.

A **first** area of computer science education that relates directly to constructivism is the search for active learning techniques. There are many types of active learning techniques, such as role-play, active story-telling, workshops, ... A comprehensive overview of such approaches is presented in the educational patterns project (Bergin *et al.*, 2001a,b).

For example, in Jiménez-Diáz *et al.* (2005), the authors present ViRPlay as a tool that enables role-play. The goal of the tool is to enable students to understand the message passing mechanism. This is important because of the difficulties students face in understanding the dynamic aspects of software (Ragonis & Ben-Ari, 2005). In ViRPlay, users get to play an object at runtime, and can react to messages being passed around. Method call, method return and object creation are represented by throwing balls around between objects containing the parameters.

A **second** area of computer science education research related to constructivism is the development of microworlds. As we already mentioned, one of the primary authors of constructionism, Seymour Papert, is responsible for the development of the LOGO programming language (Papert & Solomon, 1971; Rubinstein, 1975; Solomon, 1978; Papert, 1985). LOGO is programming language specifically targeting younger children that want to start learning computer programming. In the original LOGO system, the language was used to control mechanical devices resembling turtles. The turtles were fairly evolved *creatures*, that were capable not only of receiving commands from a computer, but also of providing feedback through various sensors. The turtles contained a pen that could be lowered to create a trace of the turtle's path, enabling the creation of all sorts of figures through the LOGO programming language.

The language is actually really simple, using a few primitives such as FORWARD, LEFT and RIGHT, to move the turtle forward and to turn left and right. The language allowed the definition of procedures composed of these primitives. For instance the code:

```
TO DRAW:DISTANCE
1 FORWARD:DISTANCE
2 BACK:DISTANCE
END
```

defines a function DRAW that takes a parameter DISTANCE that would make the turtle advance the provided distance and return to the starting point. If the pen is in the lowered state, this would draw a simple line. The simplicity of the language makes it suitable for children, who - despite of this simplicity - learn the basics of programming. The attractiveness of the language is that it invites children to play

Nr.	Level	Explanation
1	No Viewing	The absence of visualization.
2	Viewing	The core form of engagement, since all other forms of engagement with visualization technology fundamentally entail some kind of viewing. The most passive form of engagement.
3	Responding	Answering questions concerning the visualization presented by the system.
4	Changing	Modifying the visualization, e.g. by allowing the learner to change the input of the algorithm under study in order to explore the algorithms behavior in different cases.
5	Constructing	Generation of visualization through direct generation or hand generation.
6	Presenting	Presenting a visualization to an audience for feedback and discussion.

Table 3
Levels of visualization engagement (Naps *et al.*, 2002).

Levels	Description
Viewing vs. No Viewing	Viewing results in equivalent learning outcomes to no visualization (and thus no viewing). Several studies (Anderson & Naps, 2001; Byrne *et al.*, 1999; Lawrence, 1993) have shown that mere passive viewing provides no significant improvement over no visualization, but these studies were based on small sample.
Responding vs. Viewing	Responding results in significantly better learning outcomes than viewing Bridgeman *et al.* (2000); Byrne *et al.* (1999); Faltin (2002); Hundhausen (1998); Jarc *et al.* (2000); Naps *et al.* (2000).
Changing vs. Responding	Changing results in significantly better learning outcomes than responding (Anderson & Naps, 2001; Gloor, 1998; Khuri, 2001; Naps, 2001).
Constructing vs. Changing	Constructing results in significantly better learning outcomes than changing (Anderson & Naps, 2001; Hundhausen, 1998; Rodger, 1996; Stasko, 1997).
Presenting vs. Constructing	Presenting results in significantly better learning outcomes than constructing (Naps references no literature for this conclusion)

Table 4
Higher levels of engagement lead to higher performance of a visualization.

and experiment with programming languages. Creating programs that move robots is a lot more exciting than creating mathematical algorithms, certainly for novice programmers.

The **third** area of computer science education that is influenced by constructivism is that of visualization. The main idea is presented in Naps *et al.* (2002), where the authors explore the role of engagement in visualization. In their engagement taxonomy, the authors present different levels of visualization interactivity, claiming that higher levels lead to better performance. The taxonomy is represented in table 3.

According to constructivist theory, one would expect a higher level of engagement to result in a higher level of effectiveness for a visualization. The authors present a literature overview that comes to this very conclusion. Table 4 presents the results of this overview.

Constructivism teaches us that just showing visualizations is not enough for deeper learning to occur. With respect to creating visualizations, rather than just watching them, Ross (1991) compares programmers to artisans. He states that *all artisans are intrigued by what they create, and they like to observe their work from all angles, contemplating the results and pondering what might have been done better.*

Smith & Webb (2000) note that *people seem to enjoy watching how their creations work. The response was almost universal that they liked watching the way that*

the variables changed value as the program executed. Knuth (1989) mentions that he used a program animator just for the fun of watching his TEX creation. Clearly, there is an intrinsic motivation in being able to see your own work. This would also imply that visualization tools where you are creating the visualization yourself provide a higher level of motivation than pre-made visualizations (see e.g. Naps *et al.* (2002). This statement is supported by the evidence collected in Ebel & Ben-Ari (2006). In their video analysis of classroom activity for behavior patterns that are associated with uncooperative attitude and attention loss, they discovered a near-total reduction in "bad" behavior while using activating program visualization techniques.

2.3 Summary

This section has surveyed the current status quo in learning theory, identifying and introducing the two most important paradigms - cognitivism and constructivism and their impact on computer science education literature. From our discussion, it becomes clear that both cognitivism and constructivism can contribute to computer science education in important ways. Cognitivism teaches us about schema as the primary means of creating and storing knowledge. If programming is proving as difficult as it is (as we described in the first section), the difficulty of creating schema of object oriented language concepts might be the culprit. Constructivism in general, and constructionism in particular, teaches us that students that are actively involved with their subject through the creation of artifacts, learn better than students that passively absorb knowledge. We will use these two high-level guidelines as the basis for two general requirements for our CS1 tool in the next section.

3 Learning theory and CS1 tools

The previous section discussed two paradigms of learning, cognitivism and constructivism. We have translated the two paradigms of learning into two high level criteria for the development of an educational tool that can help our students understand object-oriented programming:

General criterium 1 *A CS1 tool should help in building schema of basic language concepts.*

Understanding of concepts is the first of the cognitive accomplishments from computer programming according to Dalbey & Linn (1986). Based on a review of several empirical studies, the authors describe the chain of cognitive accomplishments as follows (slightly rephrased for purposes of readability):

(i) **Language features:** in order to use a programming language, it is important to understand many of the language features or nondecomposable elements of the language. [...] Students' knowledge of language features is often assessed by comprehension items asking them to predict how programs using the features will perform. [...] Students need to learn language features. However, such knowledge is of little general (i.e. beyond the particular programming language) use or benefit.

(ii) **Design skills:** design skills are the group of techniques used to combine language features to form a program that solves a problem. These include templates and procedural skills.

(iii) **Problem-solving skills:** The third link on the chain consists of problem-solving skills useful for learning new formal systems. [...] These problem solving skills may be acquired when students attempt to apply templates or procedural skills learned in one system to a new system.

Because becoming an expert programmer takes a long time, over 10 years according to Winslow (1996), it seems wise for a CS1 course to focus on the first step of the cognitive accomplishments ladder: understanding of language features. When we state that we wish to focus on the understanding of language constructs, we are actually saying that we want a tool to help in the creation of schema of these language constructs. In the previous chapter we have presented learning as the creation of schema that can be stored in long term memory - a definition that follows from the architecture of human cognition. Schema are the mechanisms that allow us to reduce cognitive load and hence comprehend more difficult problems or perform more complicated tasks without overloading working memory. The list of schema we want students to generate is of course dependent on the content of the course.

General criterium 2 *A CS1 tool should allow for active manipulation of course content.*

This requirement is a translation of the specific type of instructional design as advocated by constructivism, described in section 2.2. More specifically, given the previous requirements, a CS1 tool should assist students in the generation and manipulation of Java concepts and thus help them in forming schema of the concepts.

To fulfill these two requirements we have opted for the development of a program visualization tool. While other types of tools would be certainly be possible and can also help in teaching object-orientation to novice programmers, the choice for a program visualization tool has been made with good cause. The next paragraphs will explain our line of reasoning. After a thorough literature review of available CS1 tools, we have identified the following broad categories:

- Algorithm visualization tools.
- Program visualization tools.
- Educational programming environments.
- Microworlds.

Important to note (as one can see from our choice of categories) is that most tools use some form of visualization. For algorithm visualization and program visualization this is obvious, but most educational programming environments and microworlds are also very graphic in nature. So, in some way, the categories are all dependent on visualization.

3.1 Algorithm and program visualization tools

- The difference between algorithm and program visualization -

The distinction between algorithm visualization and program visualization is a classic distinction in visualization literature. For instance, the taxonomy of Price *et al.* (1993), which is arguably the most used taxonomy, specifically mentions this distinction. This is what the authors state about the two types of visualization content:

> *Is the system designed to produce algorithm or program visualization? The differentiation is subtle and can best be described from a user perspective: if the system is designed to educate the user about a general algorithm, it falls into the class of algorithm visualization. If, however, the system is teaching the user about one particular implementation of an algorithm, it is more likely program visualization. Signs that the line from algorithm visualization to program visualization has been crossed include displays of program code listings as opposed to higher-level abstract code diagrams, and labeled displays of the values of particular variables, as opposed to generic data displays.*

Korhonen (2003) provides the following description:

> *Software visualization (SV) has been polarized toward two opposite domains. In one domain that we call program visualization (PV), views of program structures are generated automatically. These types of views, which refer to "debugging" of an algorithm by tracing its execution step by step, are generic, low-level views and not expressive enough to convey adequately how an algorithm functions. In the second domain, called algorithm visualization (AV), we are interested in visualizing all the states of the data structures during the execution of an algorithm.*

In general, if the visualization is at level of the various basic language constructs, such as variables and their values, objects, methods etc. one speaks about program visualization. If the visualization shows the program at a higher level of abstraction, without direct reference to the language constructs in the actual program, one speaks about algorithm visualization. Program visualization is most often generic, presenting every program in the same way, while algorithm visualization often depends on the type of algorithm the designer wants to visualize. For instance, a bubble sort algorithm and a binary search algorithm might benefit from different visual representations.

- Algorithm visualization tools -

The past decades have seen many examples of algorithm visualization tools. Notable examples are Tango (Stasko, 1990), JHAVÉ (Naps *et al.*, 2000), MatrixPro and TRAKLA2 (Korhonen *et al.*, 2001; Korhonen, 2003), and Alvis Live! (Hundhausen & Brown, 2008). Figure 2 presents a screenshot of the TRAKLA system. The goal of this system is similar to that of the other algorithm visualization systems: to provide students with a suitable graphic of the algorithm or data structure in order to enhance schema creation. Most algorithm animation systems allow students to provide their own data set for the algorithm in order to enhance interactivity.

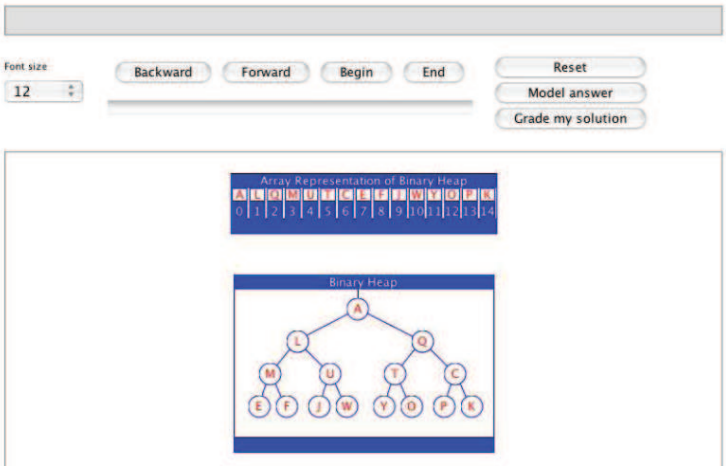

Fig. 2. An algorithm visualization in the TRAKLA2 system (Korhonen, 2003).

- Program visualization tools -

Program visualization, either static or dynamic, is closely related to the concept of the *notional machine* (du Boulay, 1996). The authors describe five problem areas for novice programmers, including understanding the notional machine. The following quotes describe this difficulty through a fitting analogy:

> *Learning to program is like learning to use a toy construction set, such as Meccano, to build mechanisms, but as if inside a darkened room with only very limited ways of seeing the innards of one's creation working. [. . .] A running program is a kind of mechanism and it takes quite a long time to learn the relation between a program on the page and the mechanism it describes. It's just as hard as trying to understand how a car engine works from a diagram in a text book. Only some familiarity with wheels, cranks, gears, bearings, etc. and "getting one's hands grubby," gives the power to imagine the working mechanism described by the diagram and crucially, to relate a broken engine's failure to work to malfunctions in its unseen innards.*

In Du Boulay *et al.* (1999) the authors expand on the idea of the notional machine. In this article, the notional machine is defined as *the idealized model of the computer implied by the constructs of the programming language.* Important in this definition is that the notional machine is language, rather than hardware dependent. Programmers, after all, work through the confines of a programming language in order to interface with the hardware. A CS1 tool should therefore present the notional machine as defined by the language, not as defined by the hardware on which programs defined in the language can run.

Further in the article the authors provide the example of prolog, where they state that *increasing the visibility of PROLOG would involve some method of dynamically displaying the stack of sub-goals and the process of scanning through the sentences of the program searching for one which matches the pattern of a sub-goal.* Figure 3 shows a visual representation of the notional machine of an Intel 8049 microcomputer software system. This representation teaches students about the inputs and outputs of this system, and about the way in which these inputs and outputs

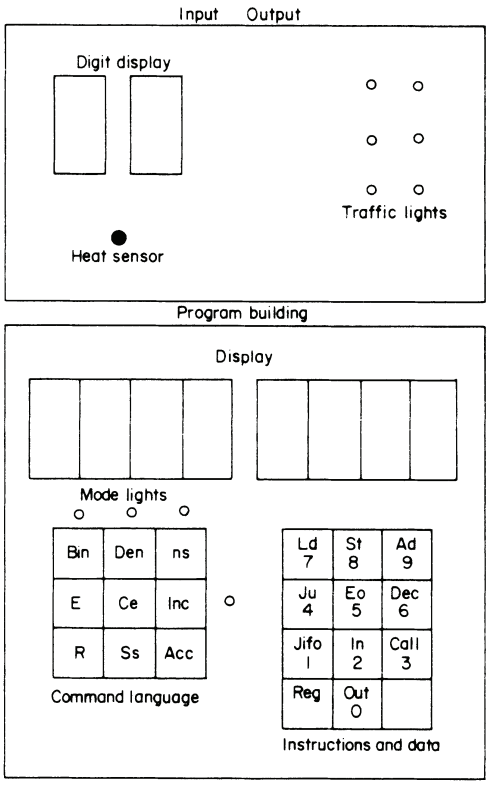

Fig. 3. The notional machine for the Intel 8049 software system (Du Boulay *et al.*, 1999).

are processed using a command language, instructions and data. The usefulness of such an explicit notional machine has long been established in other articles (see e.g. Mayer (1976)).

Similar suggestions have been provided more recently by e.g. Milne & Rowe (2002). The authors state that *the results [of their empirical study] show that the most difficult topics are so ranked because of the lack of understanding by the students of what happens in memory as their programs execute. Therefore, the students will struggle in their understanding until they gain a clear mental model of how their program is working - that is, how it is stored in memory, and how the objects in memory relate to one another.* A similar idea is presented in Schulte & Bennedsen (2006), where the authors state, as a general conclusion of their article, that *an important requirement for understanding OO programming is a good and precise understanding of the execution, and this in turn requires an understanding of the interaction between objects.*

Notable examples of program visualization systems are JIVE (Gestwicki & Jayaraman, 2005), Jeliot 3 (Moreno & Joy, 2007) and Ville (Rajala *et al.*, 2007). These tools all show the basic constructs of the Java programming language.

Jive is a visualization system that interactively shows memory state and is embedded in the Eclipse IDE as a plug-in. A screenshot of Jive in action is presented in figure 4. Gestwicki & Jayaraman (2005) present the JIVE system as follows:

A novel approach to the runtime visualization and analysis of object-oriented

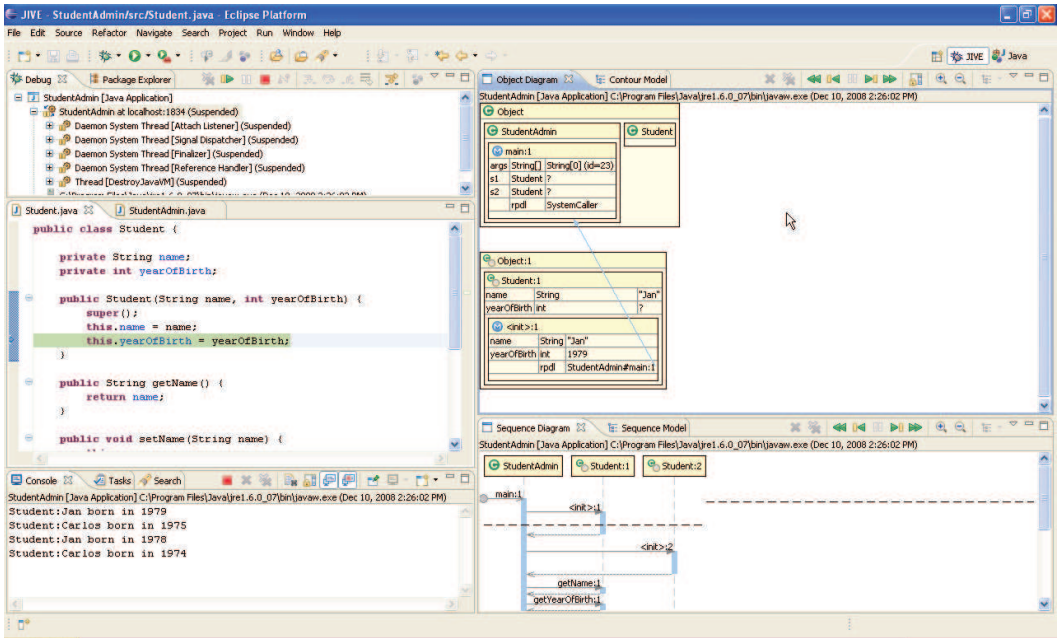

Fig. 4. The JIVE system as an Eclipse plug-in.

programs is presented and illustrated through a prototype system called JIVE: Java Interactive Visualization Environment. The main contributions of JIVE are: multiple concurrent representations of program state and execution history; support for forward and reverse execution; and graphical queries over program execution. This model facilitates program understanding and interactive debugging.

Jeliot 3 [5] is another program-level visualization tool with an integrated editor. A screenshot of Jeliot 3 in action is presented in figure 5. The authors describe Jeliot as follows (Moreno & Joy, 2007):

Jeliot 3 is a program animation tool oriented towards novice programmers, in which animations represent the step by step execution of Java programs. Each step in the execution of a program is graphically made explicit, and the resulting animation is a simulation of how the virtual machine interprets the program code. The animation takes place in a "theater" that is divided in four separated areas. The central and main area is the "Expression Evaluation" one, to which messages, method calls, values and references are moved to and from the other visualization areas as the evaluation proceeds. Students can program in Jeliot 3, and later they can visualize their program and follow its execution path.

Ville is a program visualization tool, which has the goal of offering an environment for students to study the execution of example programs whether written by students themselves or prepared by the teacher and explore the changes in the program state data structures (Rajala *et al.*, 2008). Specific about Ville is the multi-language nature of the environment. Code examples and corresponding visualization can be presented in many languages. In addition, the code execution in the different languages can be seen side by side. This has both advantages and dis-

[5] Online at http://cs.joensuu.fi/jeliot/

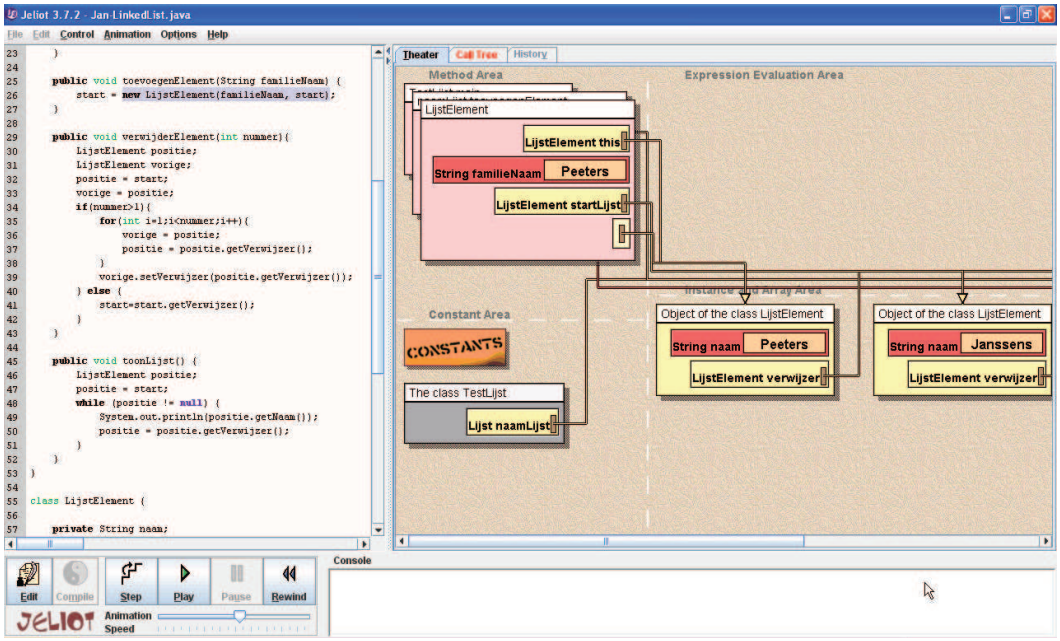

Fig. 5. Jeliot 3 in action.

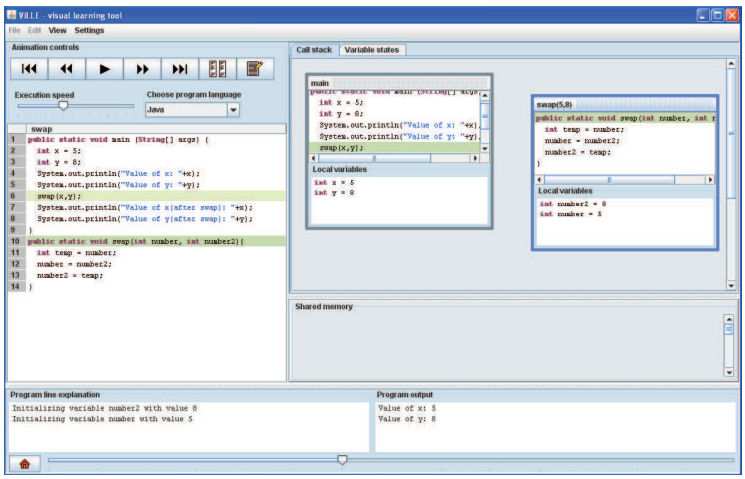

Fig. 6. Ville in action.

advantages. The advantage is that students see that many object-oriented features are language independent. The disadvantage is that only language independent features can be shown. Currently Ville supports Java and C++. Figure 6 presents a screenshot of the Ville environment.

3.2 Programming environments

An interesting field of CS1 research is devoted to the development and evaluation of Educational Integrated Development Environments or E-IDEs. Prime examples are DrJava (Allen *et al.*, 2002), BlueJ (Kölling *et al.*, 2003), ProfessorJ (Gray & Flatt, 2003) and JGrasp (Cross II & Hendrix, 2006).

The goal of DrJava is to *gently introduces students to the mechanics of writing Java programs.* The IDE uses a much simplified interface compared to full-blown

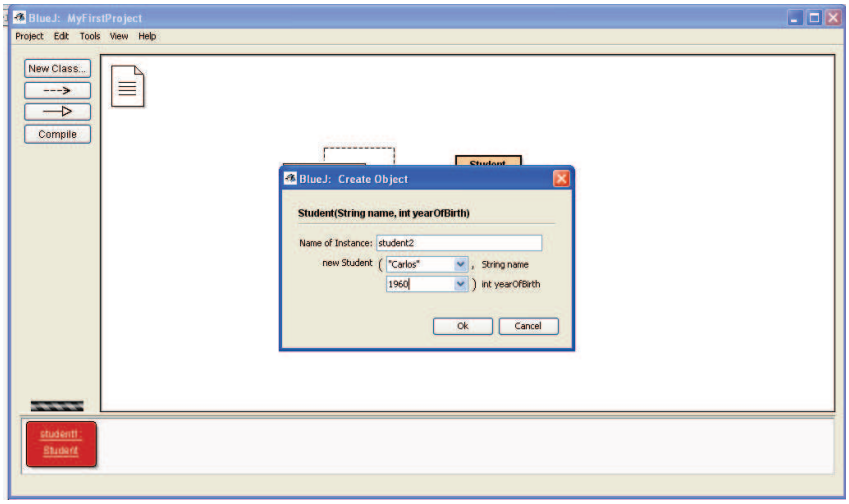

Fig. 7. BlueJ allows the creation of and interaction with objects without having to consider program flow.

IDEs. In addition, DrJava uses an implementation of DynamicJava, a freely available Java interpreter, to enable evaluation of code as the students are typing, making for a much more interactive experience. DrJava also allows students to test their code without using a main method.

BlueJ is perhaps the most cited E-IDE in literature (e.g. Hagan & Markham, 2000; Hegna & Groven, 2005; Jadud, 2006). The goal of this Java based system is to address three issues (Kölling *et al.*, 2003) of regular IDEs:

- The environment is not object-oriented.
- The environment is too complex.
- The environment focuses on user interfaces.

The BlueJ user interface consists of a UML window showing the application architecture. The user can then right-click on classes of the architecture and construct objects of the classes by graphically invoking the constructor. The created objects are placed on an object bench and can be inspected and interacted with using a GUI for calling the methods or they can be removed from the bench. This way of working removes the need for a main method and avoids the use of control flow statements or any static code for that matter, instead focusing solely on the object-oriented features of Java. BlueJ is presented in figure 7.

One of the primary features of ProfessorJ, an E-IDE based on DrScheme (Findler *et al.*, 2002), is its use of different Java language-levels, shielding students from different parts of the language based on their proficiency. These language-levels render it impossible for students to use language constructs they do not yet understand. In addition, ProfessorJ provides error messages that are meaningful to students, much in the same way as in the Expresso project (Hristova *et al.*, 2003). ProfessorJ is presented in figure 8.

JGrasp another IDE with an educational focus. The most interesting educational features of JGrasp are its visualization features, with a focus on static visualization - i.e. design time visualization. Figure 9 shows such a static visualization of a JGrasp project.

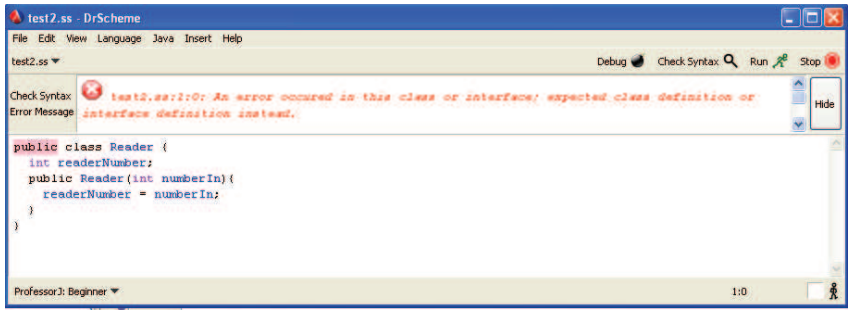

Fig. 8. The ProfessorJ environment limits the usable language constructs for novice programmers.

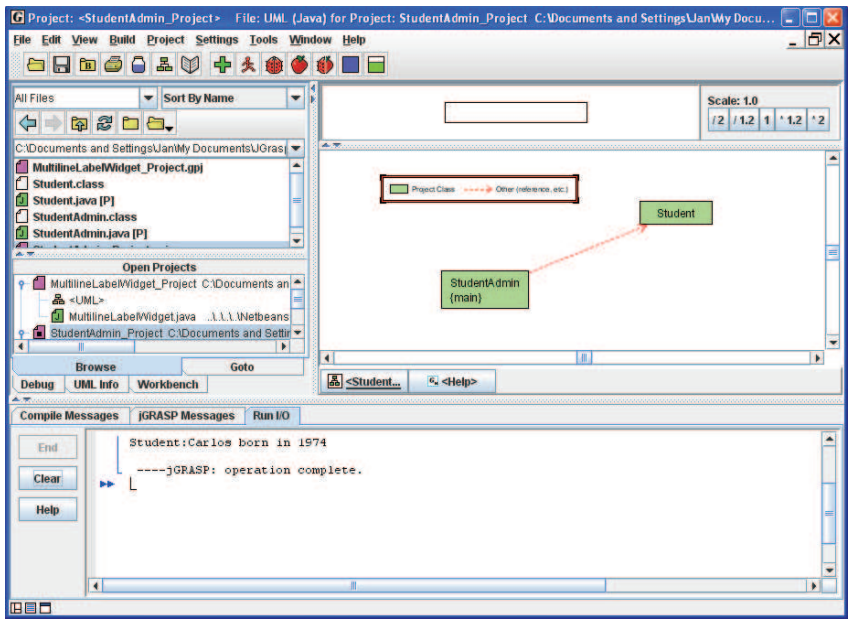

Fig. 9. The JGrasp environment showing a static visualization.

Most E-IDEs provide simplified mechanisms for creating and interacting with objects. This can be both positive and negative. On the positive side, this way of interacting (e.g. by invoking methods of a java class through a GUI, rather than through method invocation) might help in scaffolding difficult subjects. However, in the case of BlueJ, there are some reports on problems when switching to a full blown IDE in order to learn the complete Java language. Sorva & Malmi (2005) refer to Proulx *et al.* (2002) when they state that *the BlueJ environment appears to exacerbate students' difficulties with understanding the dynamics of full OO programs. [. . .] This is related to the fact that in the BlueJ-based approach [. . .] regular program entry and control flow were superseded by interactive object instantiation and method invocation within the BlueJ IDE.*

3.3 Microworlds

Microworlds often take the form of a graphical environment where the programmer can move an object, such as a robot or a kangaroo, on a canvas using a predetermined set of commands. The concept of a microworld was more or less developed by Seymour Papert, who pioneered the genre with the LOGO language and

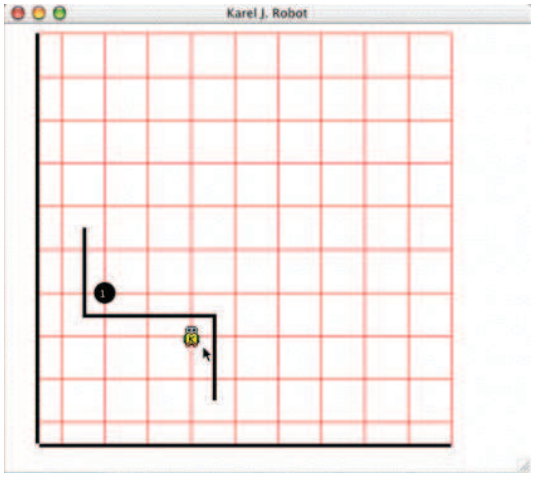

Fig. 10. Karel J. Robot in action.

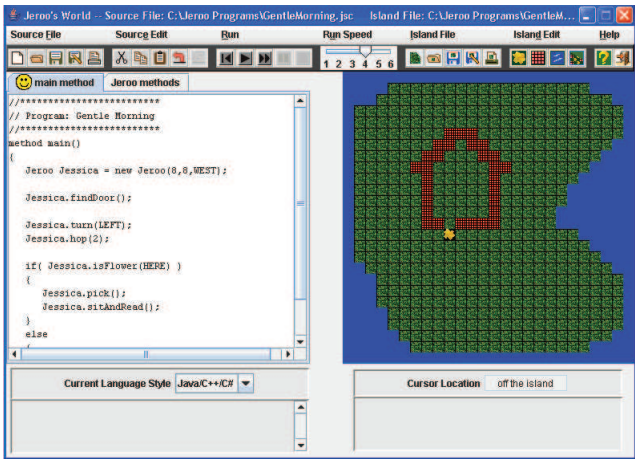

Fig. 11. Jeroo in action.

the LOGO turtles. The LOGO programming language has given rise to an entire family of microworlds. Well-known examples are Karel the Robot (Pattis, 1995), Jeroo (Sanders & Dorn, 2003b,a) and Alice (Cooper *et al.*, 2003). Also, real-world environments such as the Lego Mindstorms Kit (Holmboe *et al.*, 2004) could be considered microworlds.

The language in the original Karel environment (called Karel and very much like Pascal) was used to direct a robot through a visual environment using simple statements such as move, turnleft, turnright, ... The Karel language has been ported to C++ (Karel++) (Bergin *et al.*, 1996) and later to Java as Karel J Robot (Becker, 2001) and JKarelRobot (Buck & Stucki, 2001) and to Python as Monty Karel [6].

According to the creators, Jeroo is *similar to Karel the Robot and its descendants, but has a narrower scope than Karels descendants and has a syntax that provides a smoother transition to either Java or C++.* The goal is to have kangaroo-like animals hopping around an island picking flowers and avoiding nets. The tool uses a simplified toolkit similar to Java and allows the use of a limited set of predefined

[6] Online at http://csis.pace.edu/~bergin/MontyKarel/index.html

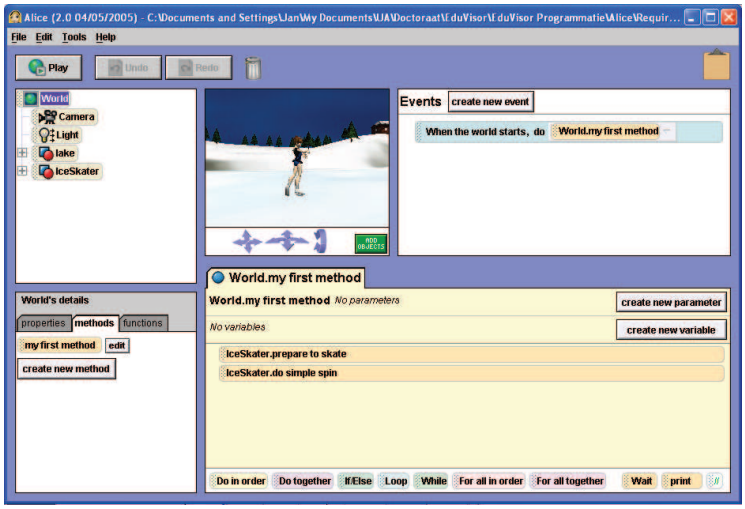

Fig. 12. Alice 3D in action.

methods. In addition, the application includes a runtime module that illustrates the connection between source code and visible actions. Jeroo is presented in figure 11.

Alice uses a 3D animated environment for representing objects. The student can add objects to the world and the microworld environment shows the objects in an object tree and in a 3D representation. Alice also provides a code editor and built-in methods conveying spatial information. A simplified GUI is used to call methods on objects in the 3D world. The goal is to create an animation using the objects in the 3D world. Alice is presented in figure 12.

Finally, the authors of Holmboe *et al.* (2004) describe how Lego can be used as a platform for learning object-oriented thinking in primary and secondary school. The same ideas are present in Barnes (2002), in which the authors present their use of the Lego Mindstorms kit in an introductory Java programming course. Using physical models, such as Lego mindstorms, instead of computer graphics, such as is the case with the Karel family, might be attractive because they provide tangible feedback to students on the workings of the programs. However, these advantages have not yet been verified.

3.4 CS1 tool category evaluation

All tools certainly may have value in a CS1 environment. However, they are not all equally viable with respect to our requirements. Algorithm visualization tools are specifically targeting algorithm and data structure visualization. They abstract away the details such as the specific methods, variables, values etc that compose the algorithm. While this may be beneficial for understanding the algorithm, the novice programmer will not benefit from this abstraction for understanding the core language concepts. Learning the schema of an algorithm should normally follow acquiring schema of language constructs, so this category fails our first requirement.

Microworlds are a very nice and gentle introduction into programming. Their simplicity make it an ideal companion for the first lessons in a CS1 course, or for teaching programming to younger children. In fact, this is how microworlds are

often used. For instance, in the introductory Java course at Stanford University [7], Karel is used in the first month, after which the class switches to Java. Microworlds indeed fall short of the goal of helping students in learning the full breadth of the Java language (or any other similar mature language) - this was never their intention. Microworlds certainly fulfill our second requirement, interactivity, but fail our first requirement, teaching schema of basic language constructs.

Educational programming environments are interesting because they can offer another approach to learning a language. They simplify the environment and they allow interaction with objects and classes without having to program. However, they also have some important disadvantages. They change the modus operandi of programming languages in a profound way, insofar that students exhibit problems when changing to a normal programming environment (see e.g. Proulx *et al.* (2002) and Ragonis & Ben-Ari (2005)). The problems with e.g. BlueJ are perhaps best illustrated by the following passage from Ragonis & Ben-Ari (2005):

> *When students were required to use separate pieces of knowledge, they did very well. They could explain the different concepts, easily operate the BlueJ environment, change given classes and even develop new classes, both simple and composed. But still they didnt have a clear picture about the execution. Nor did they clearly understand the relationships among the Java code, the BlueJ environment and the program flow. Typical expressions of these problems include: "Im mixed up. When do I need to define each of the things? What is done before what?" One very interesting question was asked when we taught the main program: "Why is it needed? We can do all these things within BlueJ."*

These specific IDEs are normally targeting objects-first courses, where objects are taught in the very beginning of the course, before the student has knowledge of control structures, methods etc. For this specific case, these IDEs are suitable because they allow the instantiation of objects without having to write a main method or constructors. However, if we want students to become confident with the language constructs the way they are normally used during programming, a detour via these specific educational IDE's might turn out detrimental (Ragonis & Ben-Ari, 2005).

This leaves us the category of program visualization. This category seems to fulfill our two requirements. The actual goal of program visualization is to provide insight into the notional machine. This means that it presents a visual picture of what is going on in memory when the program executes - making it an ideal candidate for improving schema creation. In addition, most program visualization tools allow (or even have as their primary goal) the user to create its own programs and to visualize them. This means that the user is actively involved in creating the visualization, which makes it fulfill the second goal.

[7] Online at http://www.stanford.edu/class/cs106a/

4 Design considerations for program visualization tools based on CLT, constructionism and perception theory

Having established program visualization tools as a valuable opportunity for helping teachers of object-oriented programming, we must consider how such a tool should be designed. We could for instance turn to some general design principles for visualization tools found in literature. One of the best known sets of principles is probably that of Shneiderman (1996). In the article the author proposes a "Task by datatype" taxonomy of visualization tools containing seven data types and seven tasks. The datatypes are **1-, 2-** and **3-dimensional** data, **temporal** data, **multi-dimensional** data, **tree**data and **network** data. The tasks are **overview, zoom, filter, details-on-demand, relate, history** and **extract**. The tasks are the basis of Schneiderman's *Visual Information Seeking Mantra*. The mantra goes "Overview first, zoom and filter, then details-on-demand".

We present an alternative to the design considerations of Schneiderman. Unlike Schneiderman's, our design principles are based on the theories of learning. We have identified the following design principles that apply to program visualization tools:

- Contiguity and proximity.
- Relevancy and conciseness.
- Interactivity.
- Perception-orientation.

These principles will be elaborated in the following subsections.

4.1 Contiguity and proximity

One of the first design considerations is the need for contiguity and proximity. This design consideration follows from one of the effects discussed in CLT - the split-attention effect. The effect states that if a person has to split its attention between multiple pieces of non-redundant information, cognitive load will increase. It is therefore better to integrate these different pieces of information. Figure 13 shows two different ways of presenting graphics and accompanying text to the user. Another typical example is presented in figure 14. In both cases, the integrated versions of the explanations have proved more effective for learning than the separated versions.

According to Mayer (2001), there are two types of contiguity: spatial contiguity and temporal contiguity. The spatial contiguity principle states that *students perform better [. . .] when corresponding words and pictures are presented near rather than far from each other*. The temporal contiguity principle states that *students learn better when corresponding words and pictures are presented simultaneously rather than successively*.

This effect has been demonstrated on many occasions, but the consequences have not yet been transposed to program visualization. Considering that the goal of program visualization is to present the effects of the execution of program code

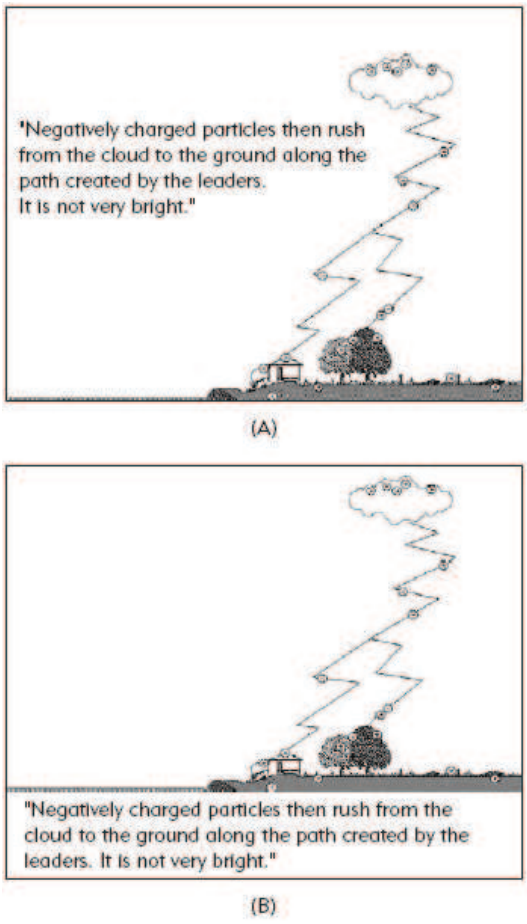

Fig. 13. Separated versus integrated graphics and text in Clark & Mayer (2007).

on the state of the execution environment, most program visualization tools show the code as well as the state of the execution environment. However, most tools that we know of maintain a clear separation between code and visualization. The only current exception that we know of is the Ville tool, but unfortunately that tool has no support for objects whatsoever, which is a clear disadvantage for an object-oriented visualization tool. Both JIVE and Jeliot have no integration between text and graphics. We propose the integration of visuals and text as an important feature for program visualization tools based on the split-attention effect discovered by Cognitive Load Theory. In our own software component, we employ the principles of contiguity and proximity in several ways:

- Method body code is located inside the method visual rather than on a separate stack (spatial contiguity).
- The method visual is located inside the current execution environment (static structure or object) (spatial contiguity).
- The method body code and the method visual are presented simultaneously (temporal contiguity).
- Objects and the classes they are derived from are automatically transported next to each other (spatial contiguity).

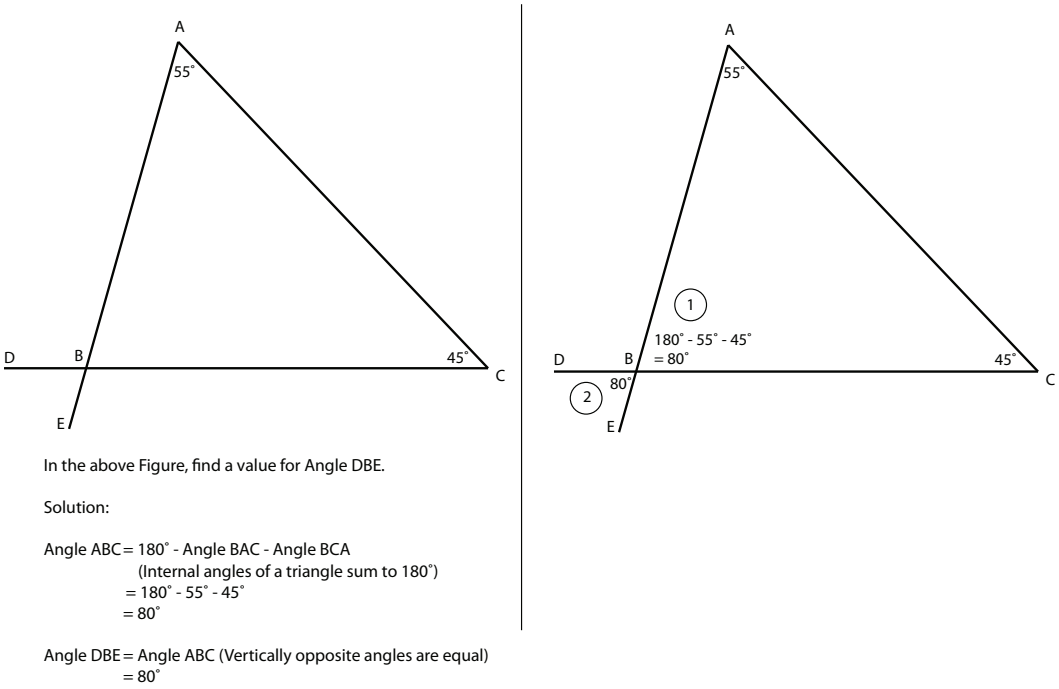

Fig. 14. Split-attention effect with graphics and text. The left example induces split-attention effect, the right example avoids split-attention (Sweller, 1998)

4.2 Relevancy and conciseness

One of the main guidelines of CLT is that the information presented to the user should always be relevant. Clark *et al.* (2006) presents three guidelines regarding relevancy:

(i) Pare content down to essentials.

(ii) Eliminate extraneous visuals, text, audio.

(iii) Eliminate redundancy.

The reason for the necessity of relevancy and conciseness is, as always, the limited capacity of our working memory. If this memory is used to process irrelevant information, less memory slots remain for processing relevant information. As with all CLT guidelines, these three are also based on empirical evidence. In Mayer *et al.* (1998), the authors compared the learning outcome of three different learning materials. One material contained a complete passage, the other contained a complete passage and a summary while the third contained only the summary. Both materials containing a summary resulted in significantly better learning outcomes than the complete passage alone.

This principle of relevancy and conciseness can be valuable for program visualization tools on many levels. Many features in our own software component are driven by this principle:

- The initial String array of a Java program, which is seldom used, can be hidden in order not to clutter the visual area.
- The user can choose whether to show all code in a running program, only the code of the currently active method or hide the code altogether, depending on

the current requirements of the user. For instance, if the user is executing a sorting algorithm running inside a method, the code of the other methods is probably not relevant, so it can be safely hidden.

- The currently active code line of a method body (which for the user is the relevant code line) is always shown with a highlight. This is true for all method frames in a running program, providing a huge advantage over traditional debuggers and other program visualization tools that only show the active codeline of the topmost method frame.
- In order to understand the duality of objects and classes, the class definition of an object in the visualization can automatically be moved right next to the visualization on mouseover, ensuring that relevant information is always adjacent on the screen. (also see section 4.1)
- The visualization component can show many relations. The relations in an inheritance hierarchy, the relations between reference variables and objects, the relations between subsequent methods on the method stack and the relations between objects and their respective classes. Each of these relationship lines can be hidden or shown based on their current relevancy.

4.3 Interactivity

As we already mentioned earlier in the article, interactivity is a requirement based on the constructivism paradigm, and is a key feature of well designed visualization systems according to Naps *et al.* (2002). The more interactive a visualization becomes, the more opportunities the user has to form schema and connections between schema in working memory.

In our own software component, interactivity is implemented in several ways:

- Our component allows the creation of animations through direct coding of the program.
- The user can request additional information on Java components by clicking the component. In this way, the visualization environment becomes more than just a program visualization tool. It becomes the center of an interactive learning environment, allowing the user to learn through interaction with the visuals.
- The user can traverse the program visualization on his own tempo, using forward, as well as backward transport buttons. The user can also make use of breakpoints in order to advance or return to a (or multiple) particular point(s) in execution history. This modus operandi results in a higher level of interactivity than what is possible using traditional virtual machines or debuggers.
- Objects on the visual scene can be moved in order to achieve a clearer visualization.
- The entire scene can be zoomed in or out.
- The user can see connections between various parts of the visualization using a simple hover technique.

Another nice example of interactivity in program visualization, not currently implemented in our own tool, is the inclusion of pop-up quizzes during program

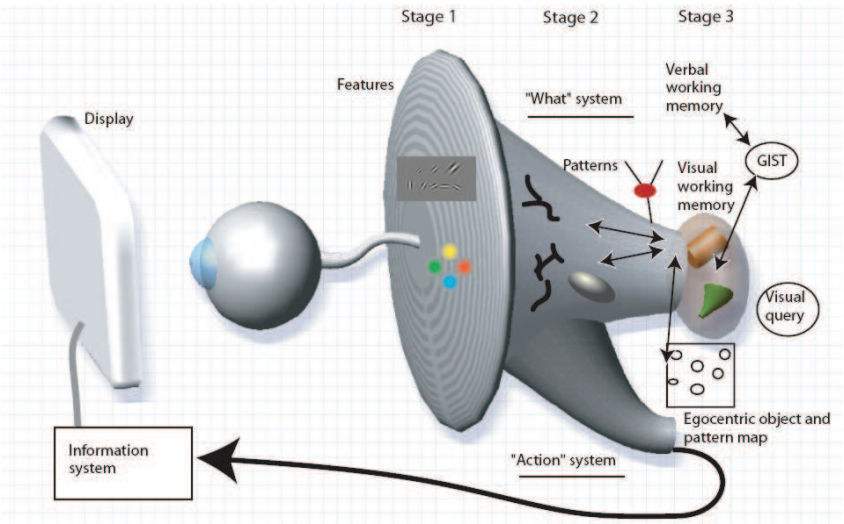

Fig. 15. The three stages of visual perception.

execution (e.g. asking for the value of a variable at certain points in time.) We have opted not to implement such features as they require the inclusion of the questions in the code - so the feature would only be useful for teachers crafting examples, not for students programming exercises. An example of such a feature can be found in the Ville Tool.[8]

4.4 Perception-orientation

The previous design principles are based on the findings of the different learning theories we have discussed in section 2. However, having established the focus of the research project to be program *visualization* software, other theories apply as well. For instance, the theory of human perception deals specifically with how we perceive (visual) information. According to Ebert (2004), *the choice of visual rendering techniques should be driven by characteristics of human perception, since perceptual channels are the communication medium.*

According to the current status quo in perception theory, there are three stages to visual perception (Ware, 2004). Figure 15 presents these three stages. In the first stage billions of neurons unconsciously scan the environment and send this information to the iconic store. The vast majority of the information in the iconic store is discarded within milliseconds. Each neuron is tuned to perceive one or a combination of features, such as color, edges, light etc. According to Ware (2004), *if we want people to understand information quickly, we should present it in such a way that it could easily be detected by these large, fast computational systems in the brain.*

In the second stage the primitive characteristics we register in the first stage are divided into regions and patterns (e.g. regions of items with the same motion or with the same color). This stage is influenced both by the massive amount of information presented through the first stage and by the *goal-directed visual queries* of the third

[8] The architecture of our component could actually be extended for this type of feature quite easily thanks to the recognition of custom annotations in the code.

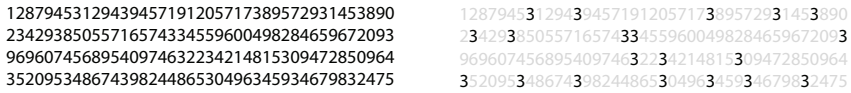

Fig. 16. Preattentive processing.

stage. The processing in this stage is much slower than first-stage processing. In the third stage we actively handle visual objects in our working memory. Capacity is extremely limited, and processing very slow.

One interesting feature of our visual system is its capability of preattentive processing (Treisman, 1982), the processing of information before we consciously pay attention to it. Preattentive processing determines what features of a visualization will most likely catch our attention, and is important because it can improve the efficiency of visual searches and thus lower cognitive load. The following features are all known to impact preattentive processing (Wolfe, 2000). Feature that have a proven influence on preattentive processing are color (hue, lightness, colorfulness), orientation, curvature, size, motion, depth cues (3D effects), Vernier offsets (relative positioning), lustre (shininess) and shape (or at least certain aspects of shape such as closure, intersections and terminators).

Of course, all of these features have been verified empirically. For instance, the use of color in human computer interaction environments was studied in Michalski & Grobelny (2008). To understand preattentive processing, we adapt an example from Ware (2004) in figure 16:

If one were to count the number of 3's in the first sequence of numbers, one would have to scan this series sequentially. If the same was asked for the second series of numbers, the preattentive nature of color (in this case the lightness of color) would make the task much easier. That is exactly the nature of a preattentive feature - it makes directing attention to specific parts of a visualization a lot easier. It is therefore striking that very few program visualization tools make use of these preattentive features to distinguish between different parts of a visualization. Our own implementation employs these preattentive features in several ways:

- Static structures, objects, arrays and compile time structures are all represented with a different color.
- Compile-time structures have square corners, run-time have rounded corners.
- Reference variables and primitive variables have different colors.
- Method stack arrows, reference arrows, arrows between objects and their classes and inheritance arrows all have a different color, in line with the color of the structures they connect.
- Motion (flicker) is used to indicate variables that change values.
- Motion is used to move classes next to the object the user is currently investigating.
- The currently executing method frame is indicated with a different hue than method frames lower on the stack.
- Thanks to the architecture of our component, it would not be difficult to allow for other codings of the visualizations (e.g. a separate hue, lightness or shape

for objects of different classes).

5 Design of the Java constructs

The design principles of the previous sections were general in nature, as they did not pertain to any particular Java construct. Of course, if the ultimate goal of the program visualization component is to present clear schema of these constructs in the hope students can more easily construct similar schema in their long term memory, the particular presentation of the constructs is just as important as the general considerations presented in the previous section. The following sections will therefore discuss each construct in detail, on a conceptual level, and deduce a schema suitable for presentation to the students. These section will reference the Java Language Specification (abbreviated to JLS) (Gosling *et al.*, 2005) and the Java Virtual Machine Specification (abbreviated as JVMS) (Lindholm & Yellin, 1999), because these are the official reference documents for the language features we aim to present. However, often the official description will be too complex to show to students, in which case we will opt for the presentation of a simplified version.

5.1 Objects

According to the JLS, *an object is a class instance or an array. The reference values (often just references) are pointers to these objects, and a special null reference, which refers to no object. There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object, and then the altered state can be observed through the reference in the other variable.*

The Java VM specification does not mandate any particular internal structure for objects or class instances. However, in Sun's own implementation of the VM, *a reference to a class instance is a pointer to a handle that is itself a pair of pointers: one to a table containing the methods of the object and a pointer to the Class object that represents the type of the object, and the other to the memory allocated from the heap for the object data.* This type of complicated structure would be far too difficult for novice students to comprehend.

Rather, we present a schema in much the same way as it is represented in many text books, or in the Unified Modeling Language object diagrams. Objects are created "on the heap" which means they are created somewhere ad random on the canvas. Every object contains the fields of the class the object belongs to, and can hold values for these fields. Method frames are added to the object when the method frames are created, and removed from the object when the frames terminate. Every object has a unique identifier, which is used to distinguish objects from each other. This identifier is used as a value for the reference variables that reference the object.

5.2 Static structures

With static structures, we mean static fields, methods and the classes these fields and methods are declared in. With regard to static fields, the JLS states that *if a field is declared static, there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A static field, sometimes called a class variable, is incarnated when the class is initialized.* Regarding static methods, the JLS states that *a method that is declared static is called a class method. A class method is always invoked without reference to a particular object.*

In our implementation, when a static field is initialized or a static method is called, the field or method appears within the visual representing the static class. These static class visuals are represented in much the same way as class instances, with these differences:

- There can only be one static structure for each class.
- A static structure does not have a uniques identifier, as it does not need to be distinguished from other static structures of the same class.
- Static structures are represented in a different color than instance structures (objects and arrays).

5.3 Variables

According to the JLS, a variable is *a storage location and has an associated type, sometimes called its compile-time type, that is either a primitive type or a reference type. A variable's value is changed by an assignment or by a prefix or postfix ++ (increment) or −− (decrement) operator.* We represent variables in accordance to the language specification. The type of every variable is mentioned, along with the name of the variable. The storage location of the variable is represented by an empty box. On initialization, the box is filled with a value which can either be of reference type or of primitive type. Most types of variables mentioned in the JLS are represented (class variables, instance variables, array components, method parameter, constructor parameters and local variables).

5.4 Arrays

According to the JVMS, *Java virtual machine arrays are also objects.* The JLS states that *an array object contains a number of variables. The number of variables may be zero, in which case the array is said to be empty. The variables contained in an array have no names; instead they are referenced by array access expressions that use nonnegative integer index values. These variables are called the components of the array. If an array has n components, we say n is the length of the array; the components of the array are referenced using integer indices from 0 to n-1, inclusive. All the components of an array have the same type, called the component type of the array. If the component type of an array is T, then the type of the array itself is written T[].*

The previous description indicates that, yes, arrays are objects, but they are a special kind of object. Arrays have array components, which are unnamed variables

that can be referenced through an index. We present arrays in a way similar to the way we present objects, but with these differences:

- At creation, arrays hold a list of unnamed variables.
- The index of each unnamed variable is presented next to the variable.
- Arrays are represented in a color different from objects.

Multi-level arrays are represented in the way they are created in the VM, as arrays of which the values of the component variables contain references to other arrays.

5.5 Methods

According to the JLS, *A method declares executable code that can be invoked, passing a fixed number of values as arguments.* When a method is executed in a running program, a *frame* of the method is created in memory. According to the JVMS, *a frame is used to store data and partial results, as well as to perform dynamic linking , return values for methods, and dispatch exceptions. [...] The sizes of the local variable array and the operand stack are determined at compile time and are supplied along with the code for the method associated with the frame. [...] Only one frame, the frame for the executing method, is active at any point in a given thread of control. This frame is referred to as the* ***current*** *frame, and its method is known as the* ***current*** *method. The class in which the current method is defined is the current class.* Frames are put on a method stack, of which there is exactly one for each executing thread in the Java Virtual Machine.

In order to keep methods and method frames comprehensible for novice programmers, we have chosen a simplified representation of these constructs. Each method frame consists of local variables and of the code associated with the method. The active code line is indicated with a highlight. When local variables become available during execution (i.e. when they are initialized), they are added to the method frame. Each method frame is located *within* the structure on which they operate (re. the contiguity and proximity principle). When the method finishes, the method is removed from the execution environment. The method stack can be shown through a chain of method stack arrows. We currently do not support multiple threads, as this is not within the scope of a CS1 course. For the schema that represents the duality between methods and method frames, we refer to section 5.6.

Figure 17 presents our design of method frames. The different aspects we have discussed are present in the figure. The frames are located inside the static structures or objects and hold currently available local variables. The current method frame is indicated in a red color (a darker color should you read this in black and white print), the background of all other frames is white. The method stack arrows connect the “main” method frame with the “toevoegenElement” method frame and the “LijstElement” method frame. In addition, the figure shows three different modes of integration of code and method frame. The first mode integrates the code of the entire method stack into the visualization (mode A), the second mode only shows the code of the active method (mode B) and the third mode hides all code in the model. The user (either the teacher or the student) can choose which mode

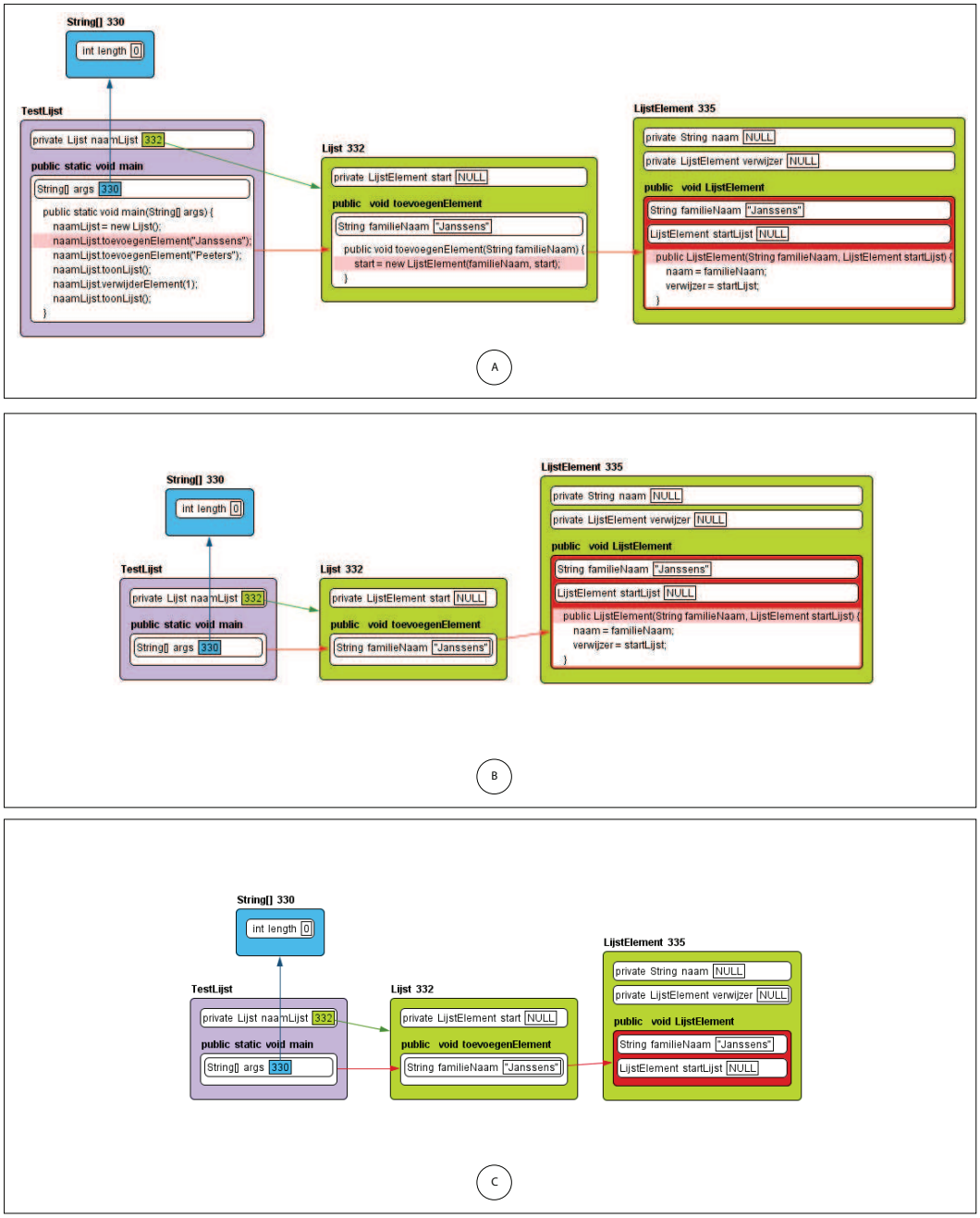

Fig. 17. Three options for integrating the code with the model.

to use based on current requirements. For instance, when an exercise contains a lengthy algorithm *inside* a method, it is probably not useful to continue showing all code in the model, as at that time only the algorithm code is important (re. the relevancy principle).

The last option assumes that the user switches to the traditional view of the code where the files are presented next to, not integrated with, the visualization (such as presented in figure 18).

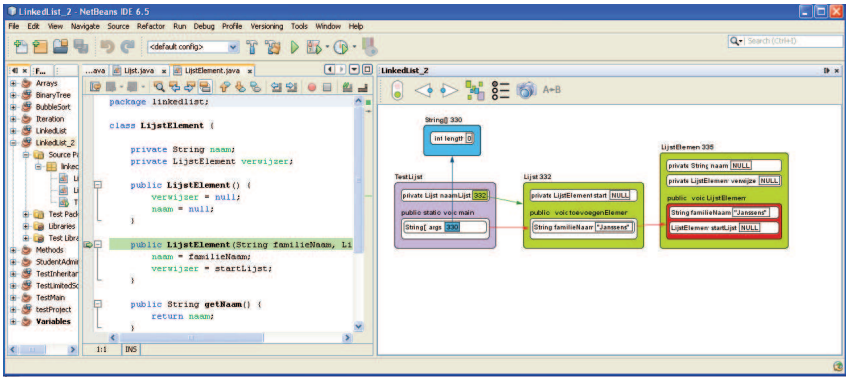

Fig. 18. The traditional approach of showing visuals and code side by side.

5.6 Compile-time structures vs. run-time structures

The distinction and interplay between objects and classes, or more generally between run-time structures and compile-time structures, lies at the heart of the inherent difficulty and, dare we say, problem, of the object oriented paradigm. The nature of this problem in fact, has been with us since the beginning of the computer science discipline. To frame this statement, we have to return to what is probably one of the most cited articles in computer science - Edsger Dijkstra's succinct but seminal case against the GOTO statement in procedural programming languages entitled *A case against the GO TO statement* (Dijkstra, 1968), later expanded upon by Wulf (1972). After describing two separate but related aspects of computer programs, i.e. the static program text on the one hand, and the dynamic program execution on the other hand, Dijkstra states the following:

> *My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitation) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.*

Dijkstra's words are a wonderful paraphrasing of the case we have been making so far, although the reasons for our incapability of understanding complex processes were not well understood at the time. Now we are a little wiser, as the body of knowledge surrounding the inner processes that govern our memory has expanded and the interactions between the various parts (perceptual buffer, working memory, long term memory) have been charted. Dijkstra then goes on to describe his reasons for denouncing the GO TO statement. The main question asked by Dijkstra is the following: *if a process is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point.* In the simplest case, if a program is just a linear sequence of statements, we just have to repeat the statements until a pointer between two successive statements in the text. Dijkstra calls such a pointer a "textual index". If conditional clauses are included, we again need just the location of one single textual index in order to reproduce behavior.

When procedures are included, Dijkstra continues, things get more complicated. In this case, when an index points to the body of a procedure, we need information

on the specific call of the procedure to determine the state of the program. The process executed by a program is now characterized by a series of indices, each index pointing to a specific procedure call. This notion of Dijkstra is akin to what is now known as the call stack, in which every frame on the stack needs its own return address.

Repetition clauses further complicate matters. Now we need a sort of dynamic index, containing the *ordinal number corresponding to the current repetition*. But even still, the process can be characterized by a unique sequence of textual and dynamic indices. The sequence of indices allows us to determine the value of a variable in the program, and thus the program state, at any point during execution. The use of the GO TO statement makes any such simple sequence (save for a complete trace of statements) impossible, as now the program can jump from any arbitrary point to any other arbitrary point in the program, exiting or entering procedures in the middle of the procedure body, possibly invalidating entire portions of the call stack. At best it becomes incredibly difficult to determine the value of a variable by considering execution history, at worst impossible.

This leads us to the difficulties introduced by object orientation. Although the rationale behind these difficulties is not completely analogous to the rationale behind the GO TO problem, object orientation also makes it incredibly difficult to infer the state of program variables by studying execution history. Completely negating the warning by Dijkstra as posited in the quote above, object orientation vastly increases the gap between the program (spread out in text space) and the process (spread out in time). In fact, because of the inherently distributed nature of object oriented programs (distributed here meaning that methods are distributed among the objects on which they act), the program text tells us virtually nothing about the values of variables at certain points during execution.

Because the gap between the program text containing the class definitions (and by extension, the compile-time structures) on the one hand, and the executing code (i.e. the run-time structures) on the other hand is very big, it is very important to show the relation between these constructs in a running program. Which leads us to the question, what exactly is the relation between the compile time structures and the runtime stuctures? The following points summarize this complicated relation.

- Every runtime structure, be it static or dynamic, is related to a single compile-time structure (in the case of inheritance, the inheritance tree is considered a single structure for simplification purposes). In other words, the structure of an object, as well as the structure of a static instance, is based on a compiled class.
- Every object of the same type, is based on just a single compiled class. In other words, there is a one to many relation between a compiled class and the objects that originate from it.
- Every object holds its own values for the fields defined in the class. In other words, while the definition of the fields is present in the compiled information, the values of the fields are proper to every separate object.
- A similar relation to the one between compiled classes and run-time objects (or static structures) is present between method compilations and method in-

stances. Every method instance (or method frame) is based on a compiled method, and at any one time the program may contain multiple instances of the same method.

- Every method instance inherits the "process" of the compiled method as described by the method code (but, in the case of Java, compiled to bytecode) and the definition of the local variables, but every method instance holds its own values for these local variables.
- However, and this is where things get really complicated, every method is *executes in the context of* a specific object or static structure, implicating that the method not only has access to the values of its own local variables, but also to the values of the fields of the object or static structure.
- In addition, every method frame also holds a program pointer (or return address) to the current location in the method process.
- At any one time, there is only one active method frame in the program, i.e. the frame at the top op the method stack (we are not considering multi-threaded application at this point).

In other words, it is very difficult to predict, or even trace, program behavior based on knowledge of the program code. To put the previous points in to Dijkstra's words, to repeat the "progress of the process" we need not only an array of indices to the specific method instances, but for every method instance we now also need an index to their environment of execution , i.e. the objects and static run-time structures, whose states are not, and can not be, known at compile time. In effect, where the array of method indices *without* object orientation presents a first level of indirection between the program state and the class definition (as the method indices only exist at runtime), the index to the object for every method frame adds another orthogonal level between the class definition and the program state.

The summary presented above does not even convey the complete picture. Things get more complicated when we start to consider the dependencies between compile-time and run-time information introduced by inheritance, interfaces, non-private scoped fields, multi-threading or aspect-oriented programming. Admittedly, not all of these subjects are featured in a beginner's course, but some, such as inheritance and interfaces, often are.

In order to convey this duality between run-time and compile-time information we have devised the visualization scheme presented in figure 19. To the best of our knowledge, no other program visualization tools try to incorporate information on the essential duality of object oriented software into the actual visualization.

Although it is difficult to convey the complete modus operandi of the visualization in a static text, the next section will try to present this part of the visualization component in greater detail. In the graphic, the square grey-background structures represent the compile-time structure of the program. Every compile-time structure (java class) contains the specification of the fields and of the methods. They also contain the method body, but this is not shown by default to save screen real-estate.

The rounded structures represent objects and static run-time structures. Every run-time structure is related to its compile-time counterpart through a grey arrow. As is indicated in the image, there can be multiple run-time structures originating

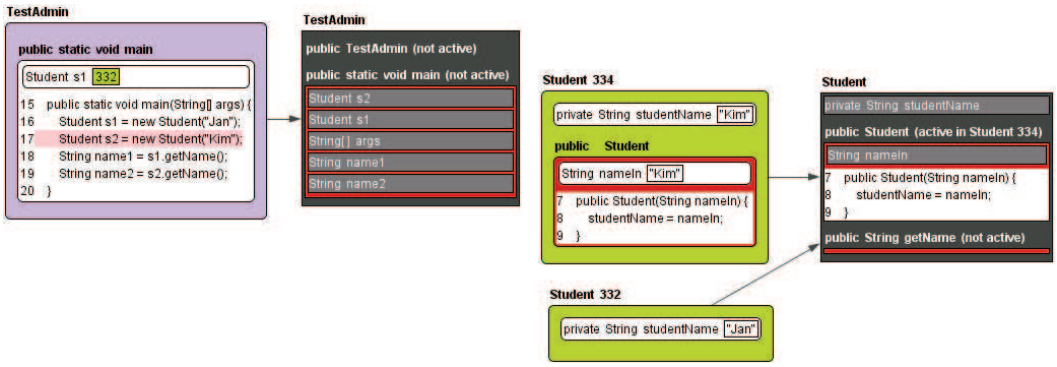

Fig. 19. Presentation of the relation between compile time and run time structures.

from the same compile-time structure. For instance, there are two Student objects originating from the same Student class. The structures of the compile-time and run-time artifacts are equal, but every run-time structure holds its own set of values. In the example, the name of the first Student object is "Kim", while the name of the second Student object is "Jan". In order to ensure proximity between relevant information and to reduce visual search, when the user hovers over a run-time structure the corresponding static structure is automatically transferred right next to said run-time structure (as a side note, most default behavior described in these sections can be changed in the configuration dialog).

The method instances are placed inside the respective run-time structures on which they operate. Again, there can be many method instances of the same method present in the program over time, or even at the same time. Every method holds it's own values for the local variables defined in the method definition. To conserve screen real-estate, the code of the methods in the compile-time structures is only shown for the active executing method. The code of the other methods is automatically hidden. In addition to revealing the method code for the active method frame in the compile-time structure, the current execution environment for the active method is also indicated in the compile-time structure using the text "active in –nameOfRunTimeStructure–". This is necessary because, as we already discussed, several method instances of the same method can be present in the method stack at any one time. Another option would have been the provision of arrows between method definitions and method instances, using a different color perhaps for the active method instance, but this quickly lead to excessive clutter of the screen.

The previous section described the relation between compile-time information and run-time information and the way it is presented in the visualization. For more information on the separate subjects (the visualization of objects, methods, variables etc.) the user is referred to their proper sections.

6 Comparison with related projects

This section presents a comparison of our own software component with similar program visualization components that are currently available. The comparison is presented in two tables, the first one dealing with what the tools are capable of visualizing, the second one dealing with the design of the visualization. In sec-

tion 3.1 we already described the primary alternatives to our program visualization component. These tools were called JEliot 3, JIVE and Ville.

Table 5 compares the tools based on the language features that can be presented to the user. The list of language features is based on an extensive review of what is often taught in CS1 courses around the world. Even though data structures and search- and sort algorithms are not part of many CS1 courses, they *are* a part of our own CS1 course (which should perhaps be better described as a CS1.5 course). In some sense, this table presents the basic requirements a tool should fulfill in order to help students form all necessary schema in an object-oriented CS1 course.

Constructs presented in the visualization	EduVizor	JIVE	Jeliot 3	Ville
Variables	√	√	√	√
Iteration	√	-	√	√
Conditionals	√	√	√	√
Methods	√	√	√	√
Arrays	√	-	√	√
Objects	√	√	√	-
References	√	√	√	-
Static vs. instance	√	√	√	-
Compile-time vs run-time	√	-	-	-
Search and sort	√	-	√	√
Data structures	√	√	√	-
Exceptions	√	-	-	-

Table 5
Comparing tools based on the features they present to the user.

Table 6 compares the tools based on the design requirements we have discussed in this article. So, in contrast to the previous table, this one deals with the *how*, not with the *what* of program visualization tools.

The comparison of the first table was one of the reasons to pursue the development of our own visualization system. There were simply no tools available that were able to present all features common in a CS1 course. In addition, our evaluation of the tools led us to identify several design issues with each of the programs. Rather than listing the issues with each of the tools separately, we have tried to provide a more comprehensive comparison based on our design criteria in this table. Of course, there are many more features that distinguish these tools apart from the design criteria, but a comparison of all features is beyond our current scope.

7 Conclusion

The difficulty of teaching programming to novice programmers is the subject of a lively research community within the computer science education research field. Many tools and methodologies have been created in order to support teachers of programming courses for novice programmers. However, many of these tools are based primarily on the common sense of the developer, rather than on the findings of established theories. This article is an attempt to provide an alternative route.

Design principle implementation	EduVizor	JIVE	Jeliot 3	Ville
Contiguity and proximity				
Method body code inside method visual	√	-	-	√
Method visual inside execution environment	√	√	-	-
Juxtaposition of run-time structure and compile-time structure	√	-	-	-
Relevancy and conciceness				
Possibility for hiding initial String array	√	-	-	-
Multiple code-hiding options	√	-	-	-
Presentation of **all** relevant code lines in program execution	√	-	-	√
Possibility of hiding irrelevant connections between constructs	√	-	-	-
Interactivity				
Creation of visualizations based on student exercices	√	√	√	√
Interactive learning environment features	√	√	√	√
Backward and forward execution	√	√	-	-
Breakpoint support	√	√	-	-
Rearranging of the visualization	√	-	-	√
Zooming	√	-	-	-
Interactive hovering reactions	√	-	-	-
Preattentive processing features				
Different colors for different structures	√	-	√	-
Different shapes for different structures	√	-	-	-
Motion for indicating changes in values	√	-	-	-

Table 6
Comparing tools based on their adherence to design principles.

We have presented a concise summary of two important paradigms of learning and how they relate to computer science in general, and to computer science education in particular. The discussion of these paradigms has led to the identification of two high-level criteria for effective educational tools, the need for the presentation of schema of language constructs and the need for interaction and creation. We have evaluated four categories of tools based on these high-level criteria. For our purposes, the category of program visualization tools seems to hold the highest promise for effectively helping students understand object-oriented programming concepts. Based on the theories of learning and the theory of perception, we have also defined four high-level design principles. For each principle we have presented the way in which the principle is applied in the design of our own program visualization tool. We have also presented schema that can be used to visualize important concepts of object-oriented programming languages.

Finally we have compared our own tool with three well known program visualization tools based on the language concepts they can show as well as on the design

criteria defined previously. While some tools come close to the amount of language features presented to the student, it is fairly obvious that the other tools make little use of the design criteria dictated by learning theory and perception theory.

References

Ackermann, E. (2001). Piaget's constructivism, papert's constructionism: What's the difference? *Future of learning group publication*.

Ahmadzadeh, M., Elliman, D. & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *SIGCSE Bull.*, **37**, 84–88.

Allen, E., Cartwright, R. & Stoler, B. (2002). Drjava: a lightweight pedagogic environment for java. *SIGCSE Bulletin*, **34**, 137–141.

Allwood, C.M. (1986). Novices on the computer: a review of the literature. *Int. J. Man-Mach. Stud.*, **25**, 633–658.

Anderson, J.M. & Naps, T.L. (2001). A context for the assessment of algorithm visualization system as pedagogical tools. In *First International Program Visualization Workshop*, 121–130, University of Joensuu Press, Porvoo, Finland.

Anderson, J.R. (1985). *Cognitive Psychology and its Implications*. Freeman, 2nd edn.

Atkinson, R.C. & Shiffrin, R.M. (1968). *The Psychology of Learning and Motivation: Advances in Research and Theory*, vol. 2, chap. Human memory: A proposed system and its control processes, 89195. Academic, Academic.

Bailie, F., Courtney, M., Murray, K., Schiaffino, R. & Tuohy, S. (2003). Objects first - does it work? *J. Comput. Small Coll.*, **19**, 303–305.

Baldwin, L. & Kuljis, J. (2000). Visualisation techniques for learning and teaching programming. In *Proc. 22nd International Conference on Information Technology Interfaces ITI 2000*, 83–90.

Barnes, D.J. (2002). Teaching introductory java through lego mindstorms models. *SIGCSE Bull.*, **34**, 147–151.

Barrows, H.S. (1996). Problem-based learning in medicine & beyond: A brief overview. *New Directions for Teaching & Learning*, **68**, 3–12.

Bateson, A., Alexander, R.A. & Murphy, M.D. (1987). Cognitive processing differences between novice and expert computer programmers. *Int. J. Man-Mach. Stud.*, **26**, 649–660.

Becker, B.W. (2001). Teaching cs1 with karel the robot in java. *SIGCSE Bull.*, **33**, 50–54.

Ben-Ari, M. (1998). Constructivism in computer science education. *SIGCSE Bulletin*, **30**, 257–261.

Ben-Ari, M. (2001). Constructivism in computer science education. *J. Comput. Math. Sci. Teach.*, **20**, 45–73.

Ben-Ari, M. (2004). Situated learning in computer science education. *Computer Science Education*, **14**, 85–100.

Bergin, J., Stehlik, M., Roberts, J. & Pattis, R.E. (1996). *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. Wiley.

Bergin, J., Eckstein, J., Manns, M.L., Sharp, H., Voelter, M., Wallingford, E. & Marquardt, K. (2001a). The pedagogical patterns project. online at http://www.pedagogicalpatterns.org/.

Bergin, J., Eckstein, J., Wallingford, E. & Manns, M.L. (2001b). Patterns for gaining different perspectives. In *8th Conference on Pattern Languages of Programs*, Illinois, USA.

Bichelmeyer, B.A. & Hsu, Y. (1999). Individually-guided education and problem-based learning: A comparison of pedagogical approaches from different epistemological views. In *Proceedings of Selected Research and Development Papers Presented at the National Convention of the Association for Educational Communications and Technology*, 9, Houston, Texas.

Boud, D. (1985). *Reflection : Turning Experience into Learning*. Routledge.

Bridgeman, S., Goodrich, M.T., Kobourov, S.G. & Tamassia, R. (2000). Pilot: an interactive tool for learning and grading. vol. 32, 139–143, ACM, New York, NY, USA.

Bruner, J. (1979). *On Knowing: Essays for the Left Hand*. Belknap Press.

Buck, D. & Stucki, D.J. (2001). Jkarelrobot: a case study in supporting levels of cognitive development in the computer science curriculum. *SIGCSE Bull.*, **33**, 16–20.

Byrne, M.D., Catrambone, R. & Stasko, J.T. (1999). Evaluating animations as student aids in learning computer algorithms. *Computers and Education*, **33**, 253–278.

Carlson, R., Chandler, P. & Sweller, J. (2003). Learning and understanding science instructional material. *Journal of Educational Psychology*, **95**, 629–640.

Chi, M.T., Feltovich, P.J. & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science*, **5**, 121–152.

Clark, R.C. & Mayer, R.E. (2007). *e-Learning and the Science of Instruction: Proven Guidelines for Consumers and Designers of Multimedia Learning*. Pfeiffer, 2nd edn.

Clark, R.C., Nguyen, F. & Sweller, J. (2006). *Efficiency in learning: Evidence-based guidelines to manage cognitive load.*. Pfeiffer.

Cooper, S., Dann, W. & Pausch, R. (2003). Teaching objects-first in introductory computer science. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, 191–195, ACM, New York, NY, USA.

Cowan, N. (2001). The magical number 4 in short-term memory: a reconsideration of mental storage capacity. *Behav Brain Sci*, **24**.

Cross II, J.H. & Hendrix, T.D. (2006). jgrasp: a lightweight ide with dynamic object viewers for cs1 and cs2. *SIGCSE Bull.*, **38**, 356–356.

Dalbey, J. & Linn, M.C. (1986). Cognitive consequences of programming: Augmentations to basic instruction. *Journal of Educational Computing Research*, **2**,

75–93.

Davies, S.P. (1990). The nature and development of programming plans. *Int. J. Man-Mach. Stud.*, **32**, 461–481.

Degrace, P. & Stahl, L.H. (1998). *Wicked Problems, Righteous Solutions: A Catolog of Modern Engineering Paradigms*. Prentice Hall PTR.

Dijkstra, E. (1968). Go to statement considered harmful. *Communications of the ACM*, **11**, 147148.

Du Boulay, B., O'Shea, T. & Monk, J. (1999). The black box inside the glass box: presenting computing concepts to novices. *Int. J. Hum.-Comput. Stud.*, **51**, 265–277.

du Boulay, J.B.H. (1996). Some difficulties of learning to program. *Journal of Educational Computing Research*, **2**, 57–73.

East, J.P. & Wallingford, E. (1997). Pattern-based programming in initial instruction (seminar). *SIGCSE Bull.*, **29**, 393.

Ebel, G. & Ben-Ari, M. (2006). Affective effects of program visualization. In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, 1–5, ACM, New York, NY, USA.

Ebert, D.S. (2004). Extending visualization to perceptualization: The importance of perception in effective communication of information. In *Visualization Handbook*, Academic Press.

Faltin, N. (2002). Structure and constraints in interactive exploratory algorithm learning. In *Revised Lectures on Software Visualization, International Seminar*, 213–226, Springer-Verlag, London, UK.

Felder, R.M. & Silverman, L.K. (1988). Learning and teaching styles in engineering education. *Engineering Education*, **78**, 67468.

Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. & Felleisen, M. (2002). Drscheme: a programming environment for scheme. *J. Funct. Program.*, **12**, 159–182.

Garner, S., Haden, P. & Robins, A. (2005). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *ACE '05: Proceedings of the 7th Australasian conference on Computing education*, 173–180, Australian Computer Society, Inc., Darlinghurst, Australia, Australia.

Gestwicki, P. & Jayaraman, B. (2005). Methodology and architecture of jive. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, 95–104, ACM, New York, NY, USA.

Gloor, P.A. (1998). User interface issues for algorithm animation. In J.T. Stasko, J.B. Domingue, M.H. Brown & B.A. Price, eds., *Software Visualization*, MIT Press.

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2005). *The Java Language Specification, 3rd edition*. Addison Wesley.

Gray, K.E. & Flatt, M. (2003). Professorj: a gradual introduction to java through language levels. In *OOPSLA '03: Companion of the 18th annual ACM SIG-*

PLAN conference on Object-oriented programming, systems, languages, and applications, 170–177, ACM, New York, NY, USA.

Hagan, D. & Markham, S. (2000). Teaching java with the bluej environment. In *Australian Society for Computers in Learning in Tertiary Education (ASCILITE 2000)*.

Harel, I. & Papert, S. (1991). *Constructionism*. Ablex Publishing.

Hegna, H. & Groven, A.K. (2005). Stumbling thru' with objects first - some observations from a study of objects first with bluej in a non-cs context. In *Proceedings of the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts (ECOOP05)*, Glasgow, UK.

Hendrix, T.D., James H. Cross, I., Maghsoodloo, S. & McKinney, M.L. (2000). Do visualizations improve program comprehensibility? experiments with control structure diagrams for java. vol. 32, 382–386, ACM, New York, NY, USA.

Holmboe, C., Borge, R. & Granerud, R. (2004). Lego as platform for learning oo thinking in primary and secondary school. In *Proceedings of the Eight Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts (ECOOP04)*, Oslo, Norway.

Hristova, M., Misra, A., Rutter, M. & Mercuri, R. (2003). Identifying and correcting java programming errors for introductory computer science students. *SIGCSE Bulletin*, **35**, 153–156.

Hundhausen, C.D. (1998). Toward effective algorithm visualization artifacts: designing for participation and negotiation in an undergraduate algorithms course. In *CHI '98: CHI 98 conference summary on Human factors in computing systems*, 54–55, ACM, New York, NY, USA.

Hundhausen, C.D. & Brown, J.L. (2008). Designing, visualizing, and discussing algorithms within a cs 1 studio experience: An empirical study. *Computers and Education*, **50**, 301–326.

Jadud, M.C. (2006). *An Exploration of Novice Compilation Behaviour in BlueJ*. Ph.D. thesis, University of Kent at Canterbury.

Jarc, D.J., Feldman, M.B. & Heller, R.S. (2000). Assessing the benefits of interactive prediction using web-based algorithm animation courseware. *SIGCSE Bull.*, **32**, 377–381.

Jeffries, R., Turner, A., Polson, P. & Atwood, M. (1981). *The processes involved in designing software. Cognitive Skills and Their Acquisition.*, 225–283. Erlbaum, Hillsdale, N.J.

Jiménez-Díáz, G., Gómez-Albarrán, M., Gómez-Martín, M.A. & González-Calero, P.A. (2005). Virplay: Playing roles to understand dynamic behavior. In *Proceedings of the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts (ECOOP05)*, Glasgow, UK.

Kessler, C.M. & Anderson, J.R. (1987). Learning flow of control: Recursive and lterative procedures. *SIGCHI Bull.*, **19**, 73–74.

Khuri, S. (2001). A user-centered approach for designing algorithm visualizations. *Informatik / Informatique, Special Issue on Visualization of Software*, 12–16.

Kim, J. & Lerch, F. (1997). Why is programming (sometimes) so difficult? programming as scientific discovery in multiple problem spaces. *Information Systems Research*, **8**, 25–50.

Kirschner, P.A., Sweller, J. & Clark, R.E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, **41**, p75 – 86.

Knuth, D.E. (1989). The errors of tex. *Softw. Pract. Exper.*, **19**, 607–685.

Kölling, M., Quig, B., Patterson, A. & Rosenberg, J. (2003). The bluej system and its pedagogy. *Computer Science Education*, **13**.

Korhonen, A. (2003). *Visual Algorithm Simulation*. Doctor of science in technology, Helsinki University of Technology.

Korhonen, A., Malmi, L. & Saikkonen, R. (2001). Matrix - concept animation and algorithm simulation system. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, 180, ACM, New York, NY, USA.

Kurland, D., Pea, R., 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**, 429457.

Larkin, J.H. & Simon, H.A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, **11**, 65–100.

Lawrence, A.W. (1993). *Empirical Studies of the Value of Algorithm Animation in Algorithm Understanding*. Ph.D. thesis, Department of Computer Science, Georgia Institute of Technology.

Lindholm, T. & Yellin, F. (1999). *Java(TM) Virtual Machine Specification*. Prentice Hall PTR, 2nd edn.

Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin*, **36**, 119–150.

Ma, L., Ferguson, J., Roper, M. & Wood, M. (2007). Improving the viability of mental models held by novice programmers. In *Proceedings of the Eleventh Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts (ECOOP07)*, Berlin, Germany.

Marcus, N., Cooper, M. & Sweller, J. (1996). Understanding instructions. *Journal of Educational Psychology*, **88**, 49–63.

Mayer, R.E. (1976). Some conditions of meaningful learning for computer programming: advance organizers and subject control of frame order. *Journal of Educational Psychology*, 143–150.

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

Mayer, R.E. (2001). *Multimedia learning*. Cambridge University Press.

Mayer, R.E., W., B., A., B., R., M. & L., T. (1998). When less is more : Meaningful learning from visual and verbal summaries of science textbook lessons. *Journal of educational psychology*, **88**, 64–73.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.D., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bulletin*, **33**, 125–180.

Michalski, R. & Grobelny, J. (2008). The role of colour preattentive processing in human-computer interaction task efficiency: A preliminary study. *International Journal of Industrial Ergonomics*, **38**, 321 – 332.

Miller, G.A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, **63**, 81–97.

Milne, I. & Rowe, G. (2002). Difficulties in learning and teaching programming - views of students and tutors. *Education and Information Technologies*, **7**, 55–66.

Moreno, A. & Joy, M.S. (2007). Jeliot 3 in a demanding educational setting. *Electronic Notes in Theoretical Computer Science*, **178**, 51 – 59, proceedings of the Fourth Program Visualization Workshop (PVW 2006).

Naps, T.L. (2001). Incorporating algorithm visualization into educational theory: A challenge for the future. *Informatik / Informatique, Special Issue on Visualization of Software*, 17–21.

Naps, T.L., Eagan, J.R. & Norton, L.L. (2000). JhavÉ—an environment to actively engage students in web-based algorithm visualizations. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, 109–113, ACM, New York, NY, USA.

Naps, T.L., Rössling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. & Ángel Velázquez-Iturbide, J. (2002). Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education*, 131–152, ACM, New York, NY, USA.

Paas, F., Tuovinen, J.E., Tabbers, H. & Van Gerven, P.W.M. (2003). Cognitive load measurement as a means to advance cognitive load theory. *Educational Psychologist*, **38**, p63 – 71.

Palmer, B., Dewar, R. & Oliver, L. (2005). On a disposition for programming. In *Proceedings of the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts (ECOOP05)*, Glasgow, UK.

Papert, S. (1980). *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA.

Papert, S. (1985). Different visions of logo. *Comput. Sch.*, **2**, 3–8.

Papert, S. & Solomon, C. (1971). Twenty things to do with a computer. AI Memos AIM-248, MIT.

Pattis, R.E. (1995). *Karel the Robot: A Gentle Introduction to the Art of Programming*. Wiley, 2nd edn.

Perkins, D., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1989). Conditions of learning in novice programmers. Tech. rep., Educational Technology Center, Cambridge, MA.

Perkins, D.N. & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, 213–229, Ablex Publishing Corp., Norwood, NJ, USA.

Petre, M. & de Quincey, E. (2006). A gentle overview of software visualisation. PPIG Newsletter.

Petre, M. & Winder, R. (1988). Issues governing the suitability of programming languages for programming tasks. In *Proceedings of the Fourth Conference of the British Computer Society on People and computers IV*, 199–215, Cambridge University Press, New York, NY, USA.

Piaget, J. (1970). *Genetic Epistemology*. Columbia University Press.

Price, B., Baecker, R. & Small, I. (1993). A principled taxonomy of sofware visualization. *Journal of Visual Languages & Computing*, **4**, 211–266.

Proulx, V.K., Raab, J. & Rasala, R. (2002). Objects from the beginning - with guis. *SIGCSE Bull.*, **34**, 65–69.

Ragonis, N. & Ben-Ari, M. (2005). On understanding the statics and dynamics of object-oriented programs. *SIGCSE Bull.*, **37**, 226–230.

Rajala, T., Laakso, M.J., Kaila, E. & Salakoski, T. (2007). Ville - a language-independent program visualization tool. In L. R. & Simon, eds., *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, Koli National Park, Finland.

Rajala, T., Laakso, M.J., Kaila, E. & Salakoski, T. (2008). Effectiveness of program visualization: A case study with the ville tool. *Journal of Information Technology Education: Innovations in Practice*, **7**.

Rodger, S.H. (1996). Integrating animations into courses. In *ITiCSE '96: Proceedings of the 1st conference on Integrating technology into computer science education*, 72–74, ACM, New York, NY, USA.

Ross, R.J. (1991). Experience with the dynamod program animator. In *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, 35–42, ACM, New York, NY, USA.

Rubinstein, R. (1975). Using logo in teaching. *SIGCUE Outlook*, **9**, 69–75.

Sanders, D. & Dorn, B. (2003a). Classroom experience with jeroo. *J. Comput. Small Coll.*, **18**, 308–316.

Sanders, D. & Dorn, B. (2003b). Jeroo: a tool for introducing object-oriented programming. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, 201–204, ACM, New York, NY, USA.

Savery, J.R. (2006). Overview of problem-based learning: Definitions and distinctions. *The Interdisciplinary Journal of Problem-based Learning*, **1**, 9–20.

Schulte, C. & Bennedsen, J. (2006). What do teachers teach in introductory pro-

gramming? In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, 17–28, ACM, New York, NY, USA.

Shneiderman, B. (1980). *Software psychology: Human factors in computer and information systems (Winthrop computer systems series)*. Winthrop Publishers.

Shneiderman, B. (1996). The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, 336, IEEE Computer Society, Washington, DC, USA.

Sjoberg, S. (2007). *International Encyclopaedia of Education 3rd Edition*, chap. Constructivism and learning. Elsevier.

Smith, P. & Webb, G. (2000). The efficacy of a low-level program visualisation tool for teaching programming concepts to novice c programmers. *Journal of Educational Computing Research*, **22**, 187–215.

Solomon, C.J. (1978). Teaching young children to program in a logo turtle computer culture. *SIGCUE Outlook*, **12**, 20–29.

Sorva, J. & Malmi, L. (2005). An object testing tool for cs1. In *Proceedings of the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts (ECOOP05)*, Glasgow, UK.

Stasko, J.T. (1990). Tango: A framework and system for algorithm animation. *SIGCHI Bulletin*, **21**, 59–60.

Stasko, J.T. (1997). Using student-built algorithm animations as learning aids. *SIGCSE Bulletin*, **29**, 25–29.

Sweller, J. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, **10**, 251–296.

Taber, K.S. (2006). Beyond constructivism: the progressive research programme into learning science. *Studies in Science Education*, **42**, 125–184.

Thompson, S.M. (2004). *An Exploratory Study of Novice Programming Experiences and Errors*. Master of science in the department of computer science, University of Victoria.

Treisman, A. (1982). Perceptual grouping and attention in visual search for features and for objects. *Journal of Experimental Psychology: Human Perception and Performance*, **8**, 194–214.

van Merrinboer, J.J.G., Kirschner, P.A. & Kester, L. (2003). Taking the load off a learners mind: Instructional design for complex learning. *Educational Psychologist*, **38**, 5–13.

Visser, W. & Hoc, J. (1990). Expert software design strategies. *Psychology of Programming*, 235–247.

von Glasersfeld, E. (1987). *The construction of knowledge Contributions to conceptual semantics.*. Intersystems Publications.

Vygotsky, L.S. (1978). *Mind in Society: Development of Higher Psychological Processes*. Harvard University Press.

Ware, C. (2004). *Information visualization, perception for design*. Morgan Kaufmann Publishers.

Weidenbeck, S. (1986). Processes in computer program comprehension. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, 48–57, Ablex Publishing Corp., Norwood, NJ, USA.

Winslow, L.E. (1996). Programming pedagogy - a psychological overview. *SIGCSE Bull.*, **28**, 17–22.

Wolfe, J.M. (2000). Visual attention. In *Seeing*, Academic Press, 2nd edn.

Wulf, W.A. (1972). A case against the goto. *SIGPLAN Not.*, **7**, 63–69.