Scenario-Based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking [*]

Giora Alexandron Michal Armoni Michal Gordon David Harel

Weizmann Institute of Science, Rehovot, 76100, Israel

ABSTRACT

We examine how students work in scenario-based and object-oriented programming (OOP) languages, and qualitatively analyze the use of abstraction through the prism of the differences between the paradigms. The findings indicate that when working in a scenario-based language, programmers think on a higher level of abstraction than when working with OOP languages. This is explained by other findings, which suggest how the declarative, incremental nature of scenario-based programming facilitates separation of concerns, and how it supports a kind of programming that allows programmers to work with a less detailed mental model of the system they develop. The findings shed light on how declarative approaches can reduce the cognitive load involved in programming, and how scenario-based programming might solve some of the difficulties involved in the use of declarative languages. This is applicable to the design of learning materials, and to the design of programming languages and tools.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Languages

Keywords

Abstraction, scenario-based programming

1. INTRODUCTION

Abstraction is a fundamental concept and core skill in computing. Just to mention a few references, Dijkstra [9] described abstraction as "the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases" (p. 864). Wing [39] referred to abstraction as one of the defining characteristics of computational thinking, and as a core skill that a computer scientist must possess. According to the task force chaired by Denning [8], abstraction is one of the three processes that characterize the discipline.

Hazzan and Kramer [25] defined the concept of abstraction in computer science (CS) and software engineering (SE) as "a cognitive means, according to which, in order to overcome complexity at a specific stage of a problem solving situation, we concentrate on the essential features of our subject of thought, and ignore irrelevant details." (p. 3). Abstraction is fundamental to many CS and SE subjects, but specifically, it is central to programming. Thus, an essential characteristic of programming languages is their approach to abstraction.

In [11], Green proposed a cognitive framework for characterizing programming languages, termed 'cognitive dimensions', which also refers to abstraction properties. We find two of its dimensions especially applicable to our analysis. The first is *abstraction gradient*, which refers to the minimum and maximum levels of abstraction, and to the readiness or desire to accept new abstractions. Languages that are *abstraction tolerant* can be used 'as they come', but also allow new abstractions to be created (C is one example). In languages that are *abstraction-hungry*, one is required to create new abstractions when starting a new program (basically this is the case for most OOP languages). The second dimension is *closeness of mapping*, which refers to the distance between the problem domain and the solution domain. The closer the distance, the easier the problem solving ought to be.

The way a programming language deals with abstraction is also tightly connected to the *programming paradigm* that underlies it. For example, procedural abstraction is about encapsulating sequences of instructions in procedures. Procedural languages separate the abstraction of control from the abstraction of data. Object-oriented languages focus on decomposing the system into objects that abstract data and control. Declarative languages concentrate on the goal that the program should achieve, and abstract away the computation that leads to it. However, as Petre and Winder [32] stated, the distinction between declarative and non-declarative programming languages is not unequivocal, but gradual. Some declarative languages totally hide the operational model that underlies them, while in others the underlying operational model is more accessible.

Scenario-based programming [21], as implemented in the visual language of *live sequence charts* (LSC) [7], is such a mid-way approach to declarative programming. It allows defining in a high-level, declarative-style, independent soft-

[*]This research was partially supported by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013). The work of the first author was supported by a grant from the Azrieli Foundation.

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591167

ware modules that are interwoven at run-time by the underlying execution tool. (We elaborate on LSC in section 2.)

Due to its importance, developing abstract thinking is a main objective of the mainstream CS and SE curricula (see [26, 27]). The common approaches to teaching abstraction in programming seem to rely on (at least) two premises: First, that abstract thinking can be developed. This means that we expect that a quality computing education will improve students' abstraction skills. The second premise is of *scaffolded learning*. This approach to learning is rooted in the ideas of Bruner [6] and in Vygotsky's concept of the *zone of proximal development* (ZPD) [37]. Basically, it means that the instruction and the learning environment should be tailored to the needs of the learners (as opposed, for example, to approaches that emphasize learning in real-life contexts), and should support a gradual development of skills.

One way or another, there is a body of work that tells us that abstraction is difficult to learn. Hazzan and Kramer [25] ascribed that to the fact that abstraction is a *soft* conceptual idea, which is never applied out of context, but is utilized when another topic is at the focus of the activity. Koppelman and van Dijk [28] blamed this on the gap between 'real-life', natural-language based problem solving and programming problem solving, which is guided by the abstraction mechanisms of the programming language. They concluded that abstraction should be taught explicitly, from the beginning, in many contexts. This recommendation might sound like a truism, but actually it is not always followed. Sivilotti and Lang [34] stated that since abstraction is such a common theme, educators and students may use it casually, without explicitly drawing attention to it. The motivation behind their work was to scaffold abstract thinking within the framework of the objects-first approach, though they mentioned that implementations of this approach might cause students to "mingle abstract and concrete state" (p. 2). To overcome this, they suggested an approach termed 'components first', which focuses on separating the concerns of the interface from the concerns of the implementation.

However, several studies claimed that object-oriented abstraction is too complicated for novices. For example, Sprague and Schahczenski [35] claimed that OOP requires a higher-level of abstraction than does procedural programming. This concurs with the findings of Haberman [13], who found that novice programmers feel comfortable with procedural abstraction, when the concept is explicitly taught. Still in the procedural framework, Haberman and Ben-David Kolikant [14] studied how a blackbox-based approach can be used for introducing basic programming concepts to novices. Obviously, the essence of the black-box concept is abstraction.

Organizing the teaching around the concept of black-box has also been suggested by other authors. In the context of OOP, examples include the work of Warford [38], and the aforementioned work of Sivilotti [34]. In the context of declarative programming, Haberman and Scherz [15] used the idea of evolving boxes to teach Abstract Data Types (ADTs). They suggested starting with the use of the ADT as black-box, gradually revealing its internal implementation. This rationale, which states that black-boxes should be taught before, or at least together with, their internal implementation, underlies the black-box oriented methods. This contrasts with the more traditional approach, which goes in the opposite direction, bottom-up.

This debate can also be seen as an instance of the more general question of whether abstraction should be taught top-down or bottom-up. Currently, it seems that the common approach is bottom-up. However, as argued in the context of black-boxes [14], or in the context of moving from low to high-level languages [2], this can lead to difficulties in using high-level abstractions. Alternatively, some recent curricula, for example the new Israeli high-school curriculum [1], emphasize the use of high-level abstractions at early stages. Emphasizing the use of high-level abstraction, especially of the kind that appears in declarative programming, is also the essence of the 'logic programming based curriculum' [33] suggested by Scherz and Haberman. They and others conducted a series of studies on the learning of Prolog by novices. All in all, their bottom-line was positive. However, other work has revealed some significant difficulties in learning Prolog. Taylor [36] found difficulties that stem from the fact that the underlying operational model was not clear to the novices. Petre [30] claimed that operational models underpin declarative reasoning (and stated that Prolog does not supply such a model). The approach described in the present paper is also high-level and declarative, but it is introduced through a language (LSC) that gives access to its underlying, lower-level model, and this seems to address some of the difficulties associated with Prolog.

Using high level abstraction also requires 'meta' and reflective thinking. Among other things, meta-cognition refers to managerial processes, such as the allocation of cognitive resources, to self-verification processes, to attitudes, etc. The contribution of this component to abstract thinking was stressed by Dijkstra [10], who emphasized (among other things) the importance of being able to control the thinking process in order to focus attention on the task at hand and ignore unnecessary details. According to Armoni [5], the 'meta' component is also important for the transfer of abstraction skills between domains.

In this study, we examine patterns of abstract thinking in LSC, which is a scenario-based language, and in Statecharts, which is an object-oriented language. Both languages are visual in nature. We analyze these patterns qualitatively in light of the kind of programming that each paradigm encourages. The findings show that with LSC students develop a less detailed mental model of the system, and work at a higher level of abstraction. We believe that this reduces the cognitive load, and that it is facilitated by the declarative, scenario-based nature of LSC.

2. LIVE SEQUENCE CHARTS AND STATECHARTS

2.1 Live Sequence Charts

The language of *live-sequence charts* (LSC) was originally introduced by Damm and Harel [7] as an extension of *message sequence charts*, and was later extended significantly in [21] and [22]. It is a visual programming language for reactive system development, and is supported by the Play-Engine [21] and PlayGo [20] programming environments (here we concentrate on the former). Below we review the three main concepts underlying LSC and Play-Engine.

2.1.1 Scenario-Based Programming and Inter-Object Specification

LSC introduces a new paradigm, termed *scenario-based programming*, and is a language that uses visual, diagrammatic syntax. The main decomposition tool that the language offers is the *scenario*. In the abstract sense, a scenario describes a series of actions that constitute a certain functionality of the system, and may include possible, necessary or forbidden actions. Since a scenario usually involves multiple objects, and emphasizes the interactions between them, scenario-based programming is inter-object by nature (see [7]). Each chart is an independent module capturing a

[1]http://www.csit.org.il/CS2012/CS-1-2-4-ver-2.6.pdf (in Hebrew)

scenario. At run-time, the execution engine interweaves the charts according to the synchronization rules. An example of an LSC is shown in Figure 1.

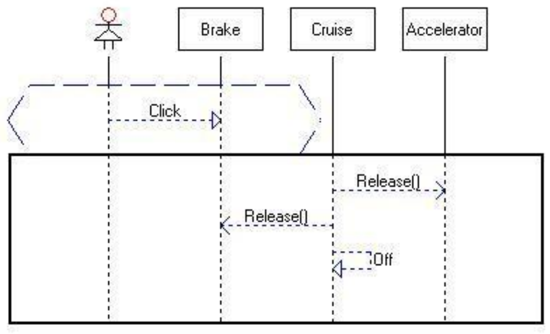

Figure 1: An LSC

An LSC is composed of two parts – the *prechart*, and the *main chart*. The prechart is the upper dashed line hexagon, and it is the activation condition of the chart. In case the events in the prechart occur, the chart is activated. Execution then enters the main chart. This is the lower rectangle in the figure, which contains the execution instructions. The vertical lines represent the objects, and the horizontal arrows represent interactions between them. Time flows from top to bottom. Figure 1 presents a simple scenario taken from the implementation of a cruise control. Once the user presses the brake pedal, the cruise unit releases control of the accelerator and the brake, and then turns itself off.

In addition to LSC, scenario-based programming is also available as an extension to Java, C++, Erlang, and Google's Blockly. The general approach has been termed *behavioral programming*. See [23].

2.1.2 The Play-In Method

LSC is supplemented with a method for building the scenario-based specification over a real or a mock-up GUI of the system – the *play-in* method [17, 21, 22] – which is implemented in the Play-Engine. With play-in, the user specifies the scenarios in a way that is close to how real interaction with the system occurs. This is illustrated in Figure 2, which shows a GUI of a cellular phone, and a simple LSC containing a scenario that describes what the display and the speaker should do once the user shuts the cell phone cover. This scenario was 'programmed' by clicking on (i.e., by *playing* with) the components of the GUI.

2.1.3 The Play-Out Method

LSC has an operational semantics that is implemented by the play-out method (originally introduced in [21, 22]). It too is included in the Play-Engine. Play-out makes the specification directly executable/simulatable, thus enables using LSC as a high-level programming language (other than merely as a specification language). For more details see [21].

2.2 Statecharts

As opposed to the inter-object approach, the more classical intra-object approach calls for specifying behavior of each object separately, usually by a state-based approach. Since OOP is based upon separating the system into objects and implementing each of the objects independently, it is intra-object by nature. An intra-object approach for specification is supported by many languages, and in particulary by the visual language of Statecharts [16], which was later adopted as part of the UML standard and is supported by various tools, such as IBM Rhapsody[2]. Rhapsody can simulate statecharts directly, or translate them into, e.g., C, C++ or Java.

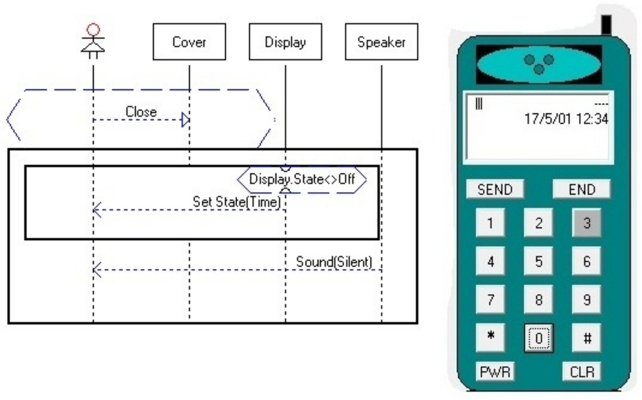

Figure 2: The play-in method

3. METHODOLOGY

3.1 Levels of Abstraction

We refer to abstraction as something that can be viewed and measured on various levels. Specifically, we use two kinds of hierarchies. Both refer to abstraction level as something that moves from the "in the large" to "in the small", as Knuth put it (quoted from Hartmanis [24], p. 39). One hierarchy refers to the level of *magnitude*, and the other hierarchy refers to the level of *meaning*. If we view information coded by bits as the lower level of abstraction, then the hierarchy of magnitude can be viewed as referring to the amount of bits that are encapsulated by (or hidden under) the abstraction means of a specific level. For example, a procedure encapsulates information into one symbol (the procedure's identifier), thus it increases the level of abstraction. The hierarchy of meaning refers to the *what* vs. the *how*. When we refer to the what, we ignore the details of the how, which are thus located on a lower level of abstraction.

Let us illustrate the use of the hierarchies with a 'real life' example. Consider the following software development scenario: i) a user creates a list of needs; ii) a system architect analyzes the list and creates a superset of groups containing needs with common characteristics; iii) the architect designs a high level architecture of the target system. Now, let us analyze the moves from i to ii, and from ii to iii, in terms of moving between levels of abstraction. When moving from i to ii, the architect analyses the user needs, so he/she still works on the same domain of what the program should do. In that domain, he/she forms an abstraction of the user needs, i.e., he/she moves up in the level of magnitude. When going from ii to iii in order to deal with the (high level design of the) implementation, he/she remains on the same level of magnitude, but moves into the domain of the how, which means a move down in the level of meaning. So i to ii is up, and ii to iii is down. But can we compare the abstraction level of i and iii? Our observation is that these activities are not comparable, because they reside on different hierarchies, and we do not have a metric for comparing a move in one hierarchy to a move in the other.

[2]http://www-01.ibm.com/software/awdtools/rhapsody/

3.2 The Research Setting

The research population was composed of two groups. The first included nine CS research students who participated in a course on “Executable Visual Languages for System Development”[3], given by the fourth-listed author in the Fall term of 2010-2011 at The Weizmann Institute of Science. The course introduced two approaches to reactive system development: The intra-object based approach of Statecharts, and the inter-object, scenario-based approach of LSC. Course assignments included implementing a system in both languages. Students’ projects included modeling electronic devices, biological systems, etc. The students were instructed to use the same system for both languages, but they could implement either the same or different parts of the system. A report on previous experience of teaching the course appears in [18].

The second group included nineteen 12th grade high-school students (age:17-18) majoring in computer science. The students participated in a mandatory course that was developed and executed as a pilot course aimed at teaching scenario-based programming and reactive system development with LSC. It was given mainly by the third-listed author. The course included theoretical and technical lectures, combined with hands-on experience in the lab. The last few lessons were devoted to developing final projects in groups of 3-5 students, under the guidance and assistance of the teacher. The task involved implementing a reactive system of the students’ choice. Projects included a memory game (‘Simon’), an elevator, etc. The students were also subjected to written exam, in which they were required to comprehend and modify LSC programs.

Hereafter we denote the group of graduate students by GR, and the high-school students by HS.

We note that the setting was not the same for both groups, as the primary goal of the two experiments was different. The main difference was that group HS was not exposed to Statechart, so we could not directly compare the two approaches in the context of this group. However, since the experience of this group included two introductory courses in Java, the students of this group did had some experience with object-based programming, which is close in nature to the intra-object approach of Statecharts.

3.3 Research Approach and Data Collection Tools

This study takes a qualitative approach, in order to develop a deep understanding of how abstraction is used in scenario-based programming vs. OOP. Data sources included post-interviews, projects, and class notes in both groups, and exam questions in group HS. In some cases we re-interpret data that we used elsewhere. The main source of data for the present study were the post-interviews. These were semi-structured, and took from thirty (HS) to sixty (GR) minutes. The focus of the interview was the LSC project, and in group GR also the Statecharts project. The students were mainly asked to describe the development process of their projects and to explain design decisions. In group GR all the students were interviewed, and in group HS four representative students only (one per group, except for one group that did not submit its project).

The analysis was composed of two main steps – defining a hierarchical model of categories, and mapping the data into the leaves of the model. The initial categories emerged from the data and led to a bottom-up development of a partial model, which was then extended in a more top-down approach. Below is a schematic representation of the model. The relation between the three high-level categories is explained in the following sections.

```
Closeness of mapping
   Similarity in structure
   Mapping between the problem and the solution
   Program comprehension
Adding a new functionality
   Using scenarios to define requirements
   Implementation first, integration later
Abstract thinking
   Working on a high level of abstraction
      Working on the level of the ‘what’
      Black-boxes and nondeterminism
         Abstract data types
         Referring to the artifact as a black-box
         Nondeterminism
      Symbolic elements
      Difficulties
   Moving between levels of abstraction
   Meta-cognitive behavior
      Awareness of abstraction levels
      Gradual refinement
      Separation of concerns
```

After the model was built, each of its leaves was associated with an operational definition (for example, one definition for ‘moving between levels of abstraction’ was ‘referring in the same sentence to different levels of abstraction’). The data was then analyzed according to the operational definitions. This was done by the first author, and was reviewed by the other authors.

The strength of qualitative research lies in its ability to study complex phenomena in depth. From that, significant insights about the phenomena can be derived, and these can guide further, more focused research. The limitations of qualitative research are its subjective nature, and the fact that (usually) it is executed in a specific context and relies on a small number of participants. These limit the generalizability of the results. Using post-factum interviews to analyze problem solving behavior, as we do, has two known limitations. First, it refers to the thinking processes in retrospect. Second, the interview itself has an effect on the interviewee’s behavior. Nevertheless, this approach is accepted in qualitative research.

4. FINDINGS

In this section, we discuss our findings. We present the analysis of groups GR and HS together, except for places in which a different behavior was observed, or places that rely on analysis that was based on data collected from only one of the groups.

The findings are arranged according to the model presented in Subsection 3.3: Subsection 4.1 refers to the high-level category of closeness of mapping. Subsection 4.2 refers to the second high-level category, which focuses on how scenario-based programming affects the way programmers add a new feature to a program that is being developed. Subsection 4.3 refers to the third high-level category, abstract thinking, and describes patterns of abstract thinking that were observed in the behavior of students working with LSC and Statecharts.

The relation between the three subsections is as follows. The first two show how the characteristics of scenario-based programming support a kind of programming that enables one, cognitive-wise, to execute several programming activities (such as program comprehension and adding a new functionality) while remaining at a higher level of abstraction. The third subsection shows that indeed this is associated with students exhibiting patterns of high-level thinking.

[3]http://www.wisdom.weizmann.ac.il/ michalk/VisLang2011/

In each subsection we describe the relevant findings, and exemplify them with a few examples. Due to lack of space, the current paper does not include the full analysis. Some of the categories were omitted, and some of the others include only a subset/summary of the findings.

4.1 Closeness of Mapping

Three aspects of closeness of mapping were found to be prominent in LSC: (1) similarity in structure, (2) mapping between the problem and the solution, and (3) program comprehension.

4.1.1 Similarity in Structure

A task in the user domain is often captured as a scenario [21]. Scenario-based programming allows one to capture a scenario in a single LSC; thus, the structure of the implementation reflects the structure of the task. Since the logic of the scenario, which is embodied in the objects and the relations between them, can be retained, the programmers do not need to invest a lot of effort in separating this logic into different modules. One result is that the cognitive effort is reduced. This is exemplified by the following excerpt. One of the students described how the scenario-based decomposition allowed her to capture a certain task in a single LSC, whereas the object-oriented decomposition forced her to invest effort in thinking about how to separate the task into objects (hereafter, students are denoted by the group prefix, followed by an identifying index):

> GR1: Basically, the implementation is pretty much the same, but there were some things that were much easier to do in Play-Engine. Things that in Statecharts you had to divide into several modules, and it was extremely unnatural. You had to think about how you implement the scenario with respect to each object, while in LSC one chart captured it all.

4.1.2 Mapping Between the Problem and the Solution

Another kind of closeness of mapping is the direct mapping between the entities of the task and the entities of the program. A task is composed of entities and the relations between them. As described above, the structure of the task can be retained by the chart that implements it. Retaining the structure seems to help in understanding the role of the objects; thus, it facilitates the mapping between the entities of the code and their counterparts in the task domain. This is exemplified by the following excerpt. A student was asked a general question about the relation between the structure of the requirements and the structure of the charts. In order to demonstrate her answer, she chose one of the LSCs, and explained what each of its entities does in terms of the scenario that it implements:

> HS1: Here, in the Extract [a name of an LSC], when you press this it shows the product that you chose [...]. Here it checks if there is a cup. This is something that the teacher told us to check for bugs [pointing to other piece of code] – if there is no cup, don't pour coffee. When you press this, it shows what you have picked.

We believe that retaining the structure helps to map between the problem and the solution. Another factor that helps in that is the high level of abstraction. We note that other studies, for example [12], have also underscored various ways in which these factors support program comprehension (which mapping is one of its aspects). One of the interesting issues that Petre [31] discussed in this context, and we briefly mention, was the Gestalt effect. According to Gestalt, the way that the human mind understands meaning in a bi-directional process – the details build the whole, but also get their meaning from it. Projecting this idea onto our domain, scenario-based programming retains the whole; thus, it facilitates understanding the meaning of the details. This is in contrast to OOP, which breaks the whole into parts in a way that loses the relationship between the elements.

4.1.3 Program Comprehension

Program comprehension is the process by which one understands the meaning of a program from its representation. The closer the distance between the level of the representation and the level of the meaning, the easier the process ought to be. Thus, comprehensibility is an indication of closeness of mapping. It can be measured, for example, by one's ability to describe what the program does in one's own words, and it seems to be an important characteristic of LSC. Among other things, it was manifested in how easy it was, even for novice programmers, to understand LSCs that were written by other students. This is demonstrated by an excerpt that was taken from a continuation of the interview from the previous section:

> I: [After the student mentioned that the work was divided among the group members] Was it easy for you to understand the things that X [a teammate] did?
> HS1: When I didn't understand I asked her, but I understood most of the things. Maybe I asked her once.

Next, the student was asked to point to a chart that her teammate had prepared. She pointed at the chart she had explained before (see the previous subsection). Whereas the latter excerpt shows that the student had a subjective feeling that she easily understood, namely, the charts written by her teammate, the former gives an objective indication that she indeed understood it. Comprehensibility was also something that the students perceived as a central advantage of LSC:

> HS4: [The main advantage of LSC is] that it is much easier to realize processes of something that causes something else. [...] If you give me something that is ready, I can understand what's happening, so later if you ask me to write what it does in my own words, I can do it.

To conclude, we believe that comprehensibility is an indication of closeness of mapping, and is supported, among other things, by the two aspects described above – the similarity in structure and the direct mapping between the problem and the solution.

4.2 Adding New Functionality

We now examine the process of adding new functionality to a program that is being developed in LSC, and we compare this to the way a similar process is executed with Statecharts. In both languages, the process of adding new functionality is done top-down from the user's domain to the program's domain, and includes two main stages:
1. Defining the new functionality in the task/user domain.
2. Adding the new functionality to the system.
We now examine how each of these stages was executed in each language.

4.2.1 Using Scenarios to Define Requirements

Stage 1 (defining the new functionality) was executed the same way in both languages. This means that we did not find that the language influenced the way the tasks were defined in the user domain. In both cases, the new functionality was defined as a use case; i.e., a scenario that describes the interaction between the system and the environment/user:

GR8: I considered the system and thought what processes can occur. The leopard is chasing them, they are running away... concrete scenarios that can occur, like I'm sitting there and watching it happen.

4.2.2 *Implementation First, Integration Later*

Stage 2 deals with the design and implementation of the new functionality. The main difference that we observed is that with LSC the students first implemented the new functionality and only later considered its integration, whereas with Statecharts it was the other way around: integration first, implementation later. This is elaborated upon below.

In LSC, the new functionality, described as a scenario, was added to the system as an LSC. Prior to implementing it, the students usually did not consider the relationship between the scenario and the existing code; i.e., they worked on the scenario without considering its integration. Only after it was implemented, did they work on its integration with the existing code. This indicates separation of concerns, though not necessarily done consciously:

I: And when you worked on these features, did you consider where the new feature synchronizes with them, like common events and things like that?
GR7: [...] basically I don't think that we considered that, we just concentrated on the new feature and how we should implement it.

This behavior illuminates one way in which the students exploited the declarative nature of LSC. Since they could rely on the underlying engine to weave the independent scenarios, they could construct their application incrementally by 'heaping' behaviors one on top of the other. With Statecharts, however, we observed a different behavior: First, the students thought about how the implementation of the new functionality could be merged into the existing code. Second, they implemented it. Of course, this behavior is a byproduct of the nature of the object-oriented paradigm: The use case, which is captured as one or more scenarios, needs to be divided according to the borderlines of the objects. Dividing the scenario between the objects means thinking about the integration, which is a task that requires planning. The following excerpts illustrate this process, and show that it can be quite exhaustive.

GR9: Another thing that we noted is that working with LSC and Play-Engine was convenient, whereas Statecharts required a lot of 'blackboard work'. We spent hours in the planning.

Later on in the interview, the student returned to this point:

I: I don't understand – was it convenient or not [in Statecharts]?
GR9: Not convenient. It was not convenient, [...] it is less obvious what is actually happening.
I: When you examine it, or also when you build it?
GR9: When you build it.
I: So what is the result? That you need to spend more time on the implementation?
GR9: Yes, exactly! In Statecharts, besides the fact that it was harder because it was the first project, there were a lot of hours spent using the blackboard. We had four or five design meetings, and two of them were blackboard meetings. In LSC, it just flowed. You just take the keyboard and start working. You see on-the-fly if it's working or not. In Statecharts you can't do anything without designing it first.

One implication is that LSC helps the programmer in separating two concerns: the implementation of a new functionality and its integration. This is further discussed in Section 5.

4.3 Abstract Thinking

In this section we refer to two aspects of abstract thinking: (1) working on a high level of abstraction, and (2) moving between levels of abstraction. We present findings that exemplify how each aspect was manifested in students' work in LSC. When possible, we compare these to findings that refer to working at this level with Statecharts or other OOP languages. Owing to space limitations, we omit the analysis of a third aspect, meta-cognitive behavior. Analysis of this sheds light on how scenario-based programming promotes meta-cognitive behavior with respect to abstraction. This level of thinking is especially important for transferring abstraction skills from one domain to another, as noted in Section 1. We intend to include this analysis in a future publication.

4.3.1 *Working at a High Level of Abstraction*

We refer to working at a high level of abstraction with respect to the two scales: meaning and magnitude. With respect to the former, we examine patterns of working at the level of what the program should do. With respect to the latter, we examine how it is manifested itself when working with black-boxes, and when using nondeterminism as a means of abstraction. We conclude by describing some difficulties that were observed.

Working at the level of 'what' – a user- vs. programmer-oriented perspective.

The highest level of abstraction is that of what the program should do, which is basically an external, user view of the system. In a previous publication [4], we showed that when working with LSC, students tended to adopt an external, usability-oriented view of the system they developed, whereas when working in an object-oriented context they tended to adopt an internal, implementation-oriented view. For example, the viewpoint was manifested in students' perception of their role with respect to the system they developed. With LSC, the programmers tended to perceive themselves as 'users', whereas with Statecharts, they tended to perceive themselves as 'programmers'.

The analysis conducted for that study was based on group GR, but it is also supported by later findings from group HS, which showed a similar pattern with respect to LSC. This is exemplified by the following excerpt:

I: And when you programmed this game, did you think like someone who is implementing this game, or like someone who is using it?
HS4: What is the difference between someone who is implementing it and someone who is using it?
I: Say, a programmer who builds it, or a user who simply plays with it.
HS4: A user, then. [...] I thought like the one who is playing, and how it would be most convenient for him.

Black-boxes and nondeterminism.

Working with a black-box means working with the interface of a functional unit without dealing with its internal implementation. Thus, a black box is regarded as high-level abstraction.

`Abstract data types:` One kind of black-box is abstract data type (ADT). Working with ADTs is inherent in the kind of programming that LSC deals with. The components of the model are implemented in one language (say, VB or

Java), and are used in LSC as black-boxes. Among other things, it seems that the fact that the components are implemented in one language and are used in a different one places a sort of conceptual 'firewall' between the two levels. This is supported by the following. Although most of the graduate students had the technical knowledge needed to work at the lower level, none of them chose to do so. In [2], we argued that the programming language gives programmers a strong cue about the level of abstraction in which they work. Thus, the fact that the black-boxes are defined in one language and are used in another one might help programmers distinguish between the two abstraction levels.

In the context of the novices, the advantage of using black-boxes to teach abstraction was discussed in Section 1. The fact that in LSC the novices reached a level of expertise that allowed them to build projects, both supports the black-box first approach and indicates that LSC offers a teaching vehicle that is appropriate for novices.

`Referring to the artifact as a black-box:` With LSC, we observed a pattern of using the artifact as a black-box during the development process. The students did not try to *anticipate* how the integrated system (the existing code + the new scenario) would behave. Instead, they simulated the system in order to *see* how it would behave. One of the experienced programmers described the simulation as a way to learn new things about the system:

> GR8: [...] It is a kind of debugging that, in the process you see that unwanted things occur, and then you update your scenarios accordingly [...]

This indicates that the student did not anticipate some of the results, but it is still possible that he/she tried to anticipate them but did not succeed in doing so. An indication that supports our conjecture that the student did not try to perceive this in advance was noted a little later in the interview:

> GR8: Things that, at the beginning, when you implement the scenario, take scenario X for example, you don't think about what happens with XYZ [examples of possible interactions of scenario X with other scenarios, and a specific example of an interaction that he/she did not anticipate].

These findings might indicate that the students had a strong perception of the system as a black-box. Since the internals of the black-box are not known, it is hard for the programmer to determine how the new code will be integrated into the existing code. This phenomenon can be also viewed from a different viewpoint. Figuring out the integration requires a mental model of the artifact. A detailed mental model enables one to mentally simulate the artifact [1]. Thus, not being able to simulate the artifact might indicate an ineffective mental model. Indeed, we observed that some of the students were not fully capable of simulating the system they developed. The pattern of using the artifact as a black-box sheds light on how programmers can exploit the declarative nature of LCS to do less work themselves, delegating hard mental tasks to the execution engine. One thing that makes this empirical evidence interesting is that this behavior is optional: LSC also allows one to work at a lower level. However, the students preferred to work at a higher level when possible, and used the lower level to underpin the process (for example, when debugging). We discuss this further in Section 5.

`Nondeterminism:` The concepts of nondeterminism (ND) and abstraction are closely related. ND allows one to ignore some of the operational details, thus it represents things more abstractly. For example, adding ND to the model of deterministic finite automaton enables one to describe state machines that are equally powerful in a more compact and abstract manner. As a nondeterministic programming language, LSC includes various nondeterministic mechanisms that facilitate abstraction. One example is its partial order semantics, which allows one to define events without committing to the order between them. In [3], we examined how LSC can be used to introduce ND to high-school students. Whereas that study focused on ND as a concept that stands on its own merit, here we examined its use from an abstraction viewpoint. Owing to space limitations, we omit more details.

Difficulties.

Some of the difficulties involved in working at a high level of abstraction are related to self-regulation and control. One needs to convince oneself that it is permissible to work at this level, and that one can ignore the irrelevant details, and not be bothered by them. In [2] we studied how previous programming experience affected the learning of new programming concepts. In the context of abstraction, we observed in some students behavioral patterns that we thought were intended to reduce the perceived abstraction level of LSC in order to deal with cognitive dissonance. In some cases, this dissonance even led to a negative attitude towards the high-level abstraction. Indeed, in [2] we conjectured that programmers tend to develop some perception of what is the 'right abstraction level', which is based on their early programming experience. The novices group was not included in that study, but the fact that in this group we did not observe such attitudes towards high-level abstraction reinforces this conjecture.

4.3.2 Moving Between Levels of Abstraction

In scenario-based programming.

In LSC, it seemed that it was relatively easy for the students to move in both hierarchies (meaning and magnitude). Among other things, this was operationalized as combining entities from different levels of abstraction in the same sentence. This is exemplified by the following excerpt, in which one of the graduate students explained the biological rationale behind one of the charts:

> GR6: In MusGluUptake [the name of an LSC], on every GluUptake – Muscle, Liver, Fat [GluUptake is a general behavior that is implemented by three different scenarios for muscles, liver, and fat]; it sends a message that it updates the Glucose level. What happens is that each of the organs updates the glucose level, because of the hormone level, etc. Actually it takes glucose, breaks down glycogen to glucose, stores the glucose, and then when it takes glucose from the blood, it reduces the glucose level. This happens with three organs.

Analyzing this through the scale of meaning shows that the student moves from the problem domain (given as the context of the question) to the solution domain (referring to the charts implementing this behavior), and then back to the problem domain (referring to the way that the organs update the glucose level). Analyzing this through the scale of magnitude can be done in both domains. In the solution domain, the student starts with a high-level description of the general behavior (GluUptake), then descends in the level of magnitude to explain how this behavior is implemented in three different places (muscle, liver, and fat), and then descends further to describe more specific details of how this is achieved. In the problem domain, the student starts with the common behavior of the organs ("each of the organs..."), then goes down to describe the specific details ("Actually it takes Glucose..."), and then makes a generalization ("This happens with three organs"), which is ascending in the abstraction level

In OOP.

In OOP, moving between levels of *meaning* seems to be more difficult. For example, a graduate student who is a very experienced programmer described how she 'forgets' about the user scenario when working according to the conventional OOP methodology, which is going from use cases to objects.

> I: So, you took some behavior, and modeled it? (referring to the work in LSC)
> GR5: Yes.
> I: And do you work the same way with object-oriented [programming languages]?
> GR5: No, only when I deal with use-cases [...].
> I: So, you think of the use-cases and then transform them into objects?
> GR5: Yes, and then when they become objects, I stop thinking about the scenarios. Maybe that's the problem...

Analyzing this through the scale of meaning shows that the student is moving from the problem domain to the solution domain, but that it is hard for her to return. We believe that this difficulty stems from the fact that OOP languages provide a less close mapping between the user's task domain and the programming entities. This is manifested, among other things, in the way that the task is divided between the objects.

Before we deal with the issue of moving between levels of magnitude in OOP, let us first examine the structure of object-oriented programs, which tends to be very layered. This is true for data – the idea behind OOP is to encapsulate everything inside objects, and this is also true for control – it is considered a good OOP habit to use small methods that delegate the 'real work' to another object. To quote Adele Goldberg, "In Smalltalk, everything happens somewhere else." (This was quoted by Nierstrasz [29], who referred to it as a general characteristic of OOP languages). The result is that tracing the flow of a 'well-written' object-oriented program requires going back and forth between methods and objects, meaning that the nature of OOP requires the programmer to constantly move between levels of abstraction. This can be very confusing, even for a programmer with several years of experience, like the one quoted below:

> GR9: In Statecharts things have a certain hierarchy, they don't work together, it's like a Matryoshka doll. The hierarchy makes it difficult to build the scenario; you become confused. It's like something is happening, then it goes to another place, to do something else... You see, in Statecharts the 'storing' [a feature of the particular system the student was working on] is built from several boxes, there is a box representing each mode, and the storing moves them from here to there.
> I: I didn't understand. Was it convenient or not [in Statecharts]?
> GR9: It was not convenient. [...] Also, when you examine it, it is less clear what is actually happening.

Nierstrasz referred to this as the "Lost in Space syndrome" of OOP [29]. Although some programmers feel very comfortable with OOP languages, for others, especially novices, the need to switch between many levels of abstraction might be too much.

5. DISCUSSION

This section is organized as follows. First, we discuss how scenario-based programming reduces the cognitive load involved in adding new functionality to an existing program. Second, we discuss how this, together with the closeness of mapping, supports programming with a less detailed mental model. Third, we show how the reduced cognitive load, the closeness of mapping, and the less detailed mental model, support working at a high level of abstraction, and the development of abstract thinking. Fourth, we discuss some possible implications of this observation to programming activities that require more concrete, less abstract thinking, such as debugging.

5.1 Reducing the Cognitive Load

5.1.1 Separation of Concerns

Scenario-based programming helps in making a clear separation between the implementation of the new functionality and its integration. As shown in Subsection 4.2.2, LSC encourages programmers to adopt a programming pattern in which they first deal with implementing the new functionality, and only later, and separately, they consider its integration.

In contrast, OOP languages seem to blur this separation. The literature and our findings suggest that this approach encourages programmers to think about the new functionality and its integration simultaneously. Since the two concerns are dealt with together, the cognitive load is consequently increased.

5.1.2 Simple First, Complex Later

Scenario-based programming encourages ordering the tasks by increasing levels of complexity: First the implementation of a new scenario, which is a local activity, then its integration, which is a broader activity that requires one to consider the relationship between various modules. OOP does the opposite. One is first required to consider the relationship between the new functionality and the existing code in order to design the solution, and only then can the more local task of implementing the behavior of each object be done.

5.1.3 Facilitating the Integration

In Subsection 4.3.1 we described the following pattern. In order to learn how an integrated program behaves, the students executed it and referred to the whole system as black box, instead of mentally simulating it. This means that the students delegated the integration task to the machine. It is somewhat similar to the difference between computing the result of a formula, and delegating the computation to a calculator. It is reasonable to assume that this behavior is cognitively easier than computing the integration, and the fact that the students chose it (probably subconsciously) is in itself evidence supporting that assumption.

5.2 Working with a Simpler Mental Model

In Subsection 4.3.1 we showed that when working with LSC, students tended to view the system as black box. We related this to findings showing that with LSC the students held a less detailed, more abstract mental model of the system. This relation is bi-directional. Not having a sufficient mental model of the artifact can lead to viewing it as black box. In contrast, when the development process can be accomplished with a less detailed mental model, there is no incentive to develop such a model (which does not mean that this is a conscious decision). We claim that scenario-based programming supports such a development process. The separation of concerns allows concentrating on the new functionality. The closeness of mapping allows it to be expressed in the language in such a way that the structure of the task is retained and one is not forced to break it up according to the structure of the solution (as in OOP). Finally, the incremental nature allows one to simulate the system with the new functionality without being concerned about how it is integrated into the previous code. We note that this analysis of the way scenario-based programming supports programming with a less detailed mental model might be applicable to other declarative programming languages that support incremental and high-level programming. However, some studies underscored the role of a clear, accessible operational model in the learning process [30, 36]. One way of resolving this apparent contradiction is by building the declarative language in a way that gives flexibility to work at both levels – the level of the declaration and the level of the execution.

5.3 Implications to Abstract Thinking

5.3.1 *Abstract Thinking Becomes Easier*

Based on the arguments discussed earlier, we contend that LSC and scenario-based programming facilitate abstract thinking and thus foster it. Since the programmer can work with a less detailed mental model, working at a high-level of abstraction becomes easier. The closeness of mapping between the problem domain and the program domain helps one move between the two representations – the level of the problem and the level of the program, which are the two main levels of abstraction. This raises a delicate issue: Closeness of mapping can lead programmers (especially novices) to confuse human discourse with formal discourse. Such confusion was reported with Prolog [36], and among other things this was related to the fact that Prolog uses terms that are taken from a natural language, which can blur the distinction between the two. With LSC, we did not note such confusion.

5.3.2 *Scaffolding*

No more, no less.

In Subsection 4.3.1 we presented some ways by which LSC and scenario-based programming allow programmers to execute several programming activities without descending to lower levels of abstraction. Among these, we mentioned the use of nondeterminism to abstract away ordering issues. This is in contrast with the deterministic nature of conventional languages, which is counter-abstraction, because it sometimes requires programmers to add unnecessary details (such as order), thus forcing them to reduce the abstraction lower than necessary.

But there is also the other side of this, which is 'too much' abstraction. Since OOP languages are "abstraction-hungry", the design in such languages usually starts with the class hierarchy. When teaching OOP, students are encouraged to design upfront a 'modular' hierarchy that will support future extensions. Since such extensions are not fully known in advance, and changing the model afterwards can be difficult, the advice is to be as abstract as possible, for example, by using abstract types. This can often lead to 'overkill', and can be exhausting, especially for novices. (Indeed, one of the motivations behind Agile Programming is that it provides a development methodology that avoids such over-engineering.) Thus, OOP languages might encourage programmers to add unnecessary levels of abstraction, which increases the cognitive load in two ways: There are more levels with which to work, and there are more levels with which to navigate.

Scenario-based programming in LSC encourages programmers to adopt a different mode of work. Abstraction is done more on a 'need to' basis. As a result, programmers tend to create abstractions that are not higher or lower than necessary. If we examine the higher level of abstraction, since the language is abstraction-tolerant, the programmers are not forced to construct a-priori abstractions that later may be found to be higher than necessary. If we examine the lower level of abstraction with respect to conventional languages, then there are several mechanisms that obviate the need to introduce unnecessary details.

Requiring vs. developing abstract thinking.

OOP does not seem to support the development of abstract thinking. It *requires* abstract thinking. OOP languages are abstraction-hungry, so one must plan the abstractions in advance, top-down, before the actual goals can be attempted [12]. Furthermore, it is not so easy to leverage the abstraction level of object-oriented programs gradually, since this might require massive refactoring. This means that when beginning to work with OOP languages, one should already possess good abstraction skills. LSC is abstraction-tolerant. One can start working with it as is, but it also accepts new abstractions easily. Also, owing to the closeness of mapping and to the 'no more/no less' effect discussed above, there is less distance between the lowest and the highest levels of abstraction. Thus, there are less levels that the programmer is required to work in and move between. The bottom line is that scenario-based programming requires less a-priori abstraction skills, and enables developing them on-the-fly. From a scaffolding point of view, this means that LSC and scenario-based programming can serve as a good tool for developing abstract thinking. This result might also apply to the behavioral programming approach that generalizes scenario-based programming (see Section 2.1.1), which we have implemented in more conventional languages, such as Java, C++ and Erlang, but this requires further study.

5.4 Mental Models and Debugging

Debugging involves comparing the program's actual behavior with the programmer's perception of what it should do. According to Adelson and Soloway [1], this process relies on simulation of the programmer's mental model. In previous sections we claimed that in languages that require a detailed mental model, the process of adding new functionality is more cognitively demanding. Thus, there might be a trade-off between the cognitive effort required when building the system and that required when debugging it: In languages that reduce the cognitive load involved in building the system by leveraging programming to a higher level, debugging might be more challenging. However, this is true for conventional debugging approaches that focus on finding the bug and fixing it in place. Scenario-based programming includes the concept of 'forbidden scenarios', which are scenarios that the system is not allowed to execute. Among other options, this can be used for non-intrusive repair of various kinds of bugs by forbidding the scenarios that lead to them, and this can be done without going 'under the hood' (for more details see [19, 21]).

6. SUMMARY AND CONCLUSIONS

In this study we investigated the relationship between the programming language (and the paradigm that underlies it) and abstract thinking. The findings indicate that when working in a scenario-based language, programmers exhibit a higher level of thinking regarding abstraction. This can be explained by other findings that show how programmers use the declarative, incremental nature of scenario-based programming in such a way that it reduces the cognitive load.

We believe that these results are applicable mainly in two domains. In the educational domain, they suggest that scenario-based languages like LSC can be used to develop abstract thinking. The feasibility of this is based on the results of a pilot semestrial course on LSC and scenario-based programming that was given to 12th grade high-school students, and on broader experience with teaching graduate students.

From a broader perspective, the results shed light on the cognitive aspects of using declarative, scenario-based, and object-oriented languages. This can be of interest for those who develop programming tools, and for those who need to choose an implementation language according to the characteristics of the developers and the task.

7. REFERENCES

[1] B. Adelson and E. Soloway. The Role of Domain Expenence in Software Design. *IEEE Trans. Softw. Eng.*, 11(11):1351–1360, Nov. 1985.

[2] G. Alexandron, M. Armoni, M. Gordon, and D. Harel. The effect of previous programming experience on the learning of scenario-based programming. Koli Calling '12, pages 151–159. ACM, 2012.

[3] G. Alexandron, M. Armoni, M. Gordon, and D. Harel. On Teaching Programming with Nondeterminism. WiPSCE '13. ACM, 2013.

[4] G. Alexandron, M. Armoni, and D. Harel. Programming with the User in Mind. *The 23rd Workshop of the Psychology of Programming Interest Group (PPIG)*, 2011.

[5] M. Armoni. On Teaching Abstraction in CS to Novices. *Journal of Computers in Mathematics and Science Teaching*, 32(3):265–284, July 2013.

[6] J. BRUNER. *The Process of Education, Revised Edition*. A Harvard paperback. Harvard University Press, 1977.

[7] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.

[8] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, Jan. 1989.

[9] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, Oct. 1972.

[10] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[11] T. R. G. Green. Cognitive dimensions of notations. In *People and Computers V*, pages 443–460. University Press, 1989.

[12] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *JOURNAL OF VISUAL LANGUAGES AND COMPUTING*, 7:131–174, 1996.

[13] B. Haberman. High-school students' attitudes regarding procedural abstraction. *Education and Information Technologies*, 9(2):131–145, June 2004.

[14] B. Haberman and Y. B.-D. Kolikant. Activating "black boxes" instead of opening "zippers" - a method of teaching novices basic CS concepts. *SIGCSE Bull.*, 33(3):41–44, June 2001.

[15] B. Haberman and Z. Scherz. Evolving boxes as flexible tools for teaching high-school students declarative and procedural aspects of logic programming. ISSEP '05, pages 156–165. Springer-Verlag, 2005.

[16] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[17] D. Harel. From Play-In Scenarios to Code: An Achievable Dream. *Computer*, 34(1):53–60, 2001.

[18] D. Harel and M. Gordon-Kiwkowitz. On Teaching Visual Formalisms. *IEEE Softw.*, 26:87–95, May 2009.

[19] D. Harel, G. Katz, A. Marron, and G. Weiss. Non-intrusive repair of reactive programs. ICECCS '12, pages 3–12. IEEE Computer Society, 2012.

[20] D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: Towards a Comprehensive Tool for Scenario Based Programming. ASE '10, pages 359–360. ACM, 2010.

[21] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[22] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. *Software and Systems Modeling (SoSyM)*, 2(2):82–107, 2003.

[23] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Commun. ACM*, 55(7):90–100, July 2012.

[24] J. Hartmanis. Turing award lecture on computational complexity and the nature of computer science. *Commun. ACM*, 37(10):37–43, Oct. 1994.

[25] O. Hazzan and J. Kramer. Abstraction in Computer Science & Software Engineering: A Pedagogical Perspective. *System Design Frontier Exclusive Frontier Coverage on System Designs*, 4(1):6–14, 2007.

[26] IEEE/ACM. Computer Science Curricula 2013.

[27] IEEE/ACM. Software Engineering 2004 – Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, 2004.

[28] H. Koppelman and B. van Dijk. Teaching abstraction in introductory courses. ITiCSE '10, pages 174–178, 2010.

[29] O. Nierstrasz. Ten Things I Hate About Object-Oriented Programming, 2010. A Banquet speech given at ECOOP.

[30] M. Petre. Shifts in Reasoning about Software and Hardware Systems: Must Operational Models Underpin Declarative Ones? *The 3rd Workshop of the Psychology of Programming Interest Group (PPIG)*, 1991.

[31] M. Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, June 1995.

[32] M. Petre and R. L. Winder. On languages, models and programming styles. *Comput. J.*, 33(2):173–180, 1990.

[33] Z. Scherz and B. Haberman. Logic programming based curriculum for high school students: the use of abstract data types. *SIGCSE Bull.*, 27(1):331–335, 1995.

[34] P. A. Sivilotti and M. Lang. Interfaces first (and foremost) with java. SIGCSE '10, pages 515–519. ACM, 2010.

[35] P. Sprague and C. Schahczenski. Abstraction the key to CS1. *J. Comput. Sci. Coll.*, 17(3):211–218, 2002.

[36] J. Taylor. Analysing novices analysing Prolog: what stories do novices tell themselves about Prolog? *Instructional Science*, 19(4-5):283–309, 1990.

[37] L. Vygotsky and M. Cole. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, 1978.

[38] J. S. Warford. Blackbox: a new object-oriented framework for cs1/cs2. *SIGCSE Bull.*, 31(1), 1999.

[39] J. M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, Mar. 2006.