Appeared as a Two-Part Article in the Journal of Object-Oriented Programming, May/June 2001

A SmallTalk for Students — A Giant Leap for Studentkind

Thomas Kühne

University of Kaiserslautern
kuehne@informatik.uni-kl.de

Abstract

We describe desirable properties of a beginner's object-oriented programming language and environment. We then compare, in an educational context, language concepts and associated environments of SMALLTALK to other more widely used languages. Building on SMALLTALK's interactive environment and exploiting the language's remarkable flexibility we produced VisualExpress, a tool that automatically produces object diagrams entirely transparent to student code. We conclude that modern constructivist approaches to education can in many respects be better realized using SMALLTALK then with typical choices such as JAVA or EIFFEL.

1 Introduction

Recent developments are causing the educational domain to change. It is becoming increasingly difficult for universities to keep a sufficient number of qualified staff due to the reduced amounts of funding available and the better paying competing industry. At the same time student numbers are increasing. Because of these forces certain pedagogical ideas have recently become very popular. Terms like learner managed learning [3] and learning contracts have almost acquired buzzword status.

These ideas may resolve the current tensions in education as they shift the emphasize from teaching to learning. The critical role of experience in learning is rediscovered and the general idea is to provide real word, case-based learning environments, rather than predetermined instructional sequences.

If it can be put to work such an approach requires less instruction and less supervision, hence, providing a solution to the current problems in education. However, for self-directed learning to work the appropriate tools, i.e., an constructivist learning environment has to be provided. Furthermore, instruction has to aim at students being able to help themselves (e.g., through the use of a debugger).

Accordingly, we describe the desirable properties of a beginner's object-oriented language in Sect. 2 and then elaborate in Sect. 3 how the language SMALLTALK and its environment fulfill the identified criteria. Section 3 also introduces the VisualExpress project, which is meant to significantly support student learning. Section 4 investigates areas where other languages are said to excel over SMALLTALK. Section 5 briefly discusses SMALLTALK's potential beyond beginner courses and we discuss related work in Sect. 6. Throughout the paper, we draw from the author's teaching experiences in order to consider typical beginner difficulties and common mistakes [18]. Section 7 concludes by summarizing the potential contributions of SMALLTALK and VisualExpress for a constructivist learning environment.

2 What is a Good Beginner's Language?

This section outlines which language and environment properties we consider important for a beginner's course.

2.1 No distraction from the main goals

In general a language should cause as little distraction from the underlying principles to be taught as possible. After all, even though a first object-oriented language has particular significance for the learner, it is only used as a vehicle to teach the concepts of classes, instances, encapsulation, polymorphism, inheritance, etc. Undoubtedly, the role of a first language is to communicate these concepts as easily as possible to facilitate the later comprehension of other object-oriented languages. If the language chosen has industrial relevance then all the better, but its primary task must be the establishment of clean concepts in the learner's mind. As a result, the language should be clean, simple, and purely object-oriented. For instance, it should not be necessary to explicitly specify if dynamic binding of method calls is required. Also, machine details such as integer overflows, basic and non-basic types, and memory management should not be exposed to students (see Sect. 2.2).

EIFFEL certainly abounds with sound concepts with a clean syntax [25]. Although it has the best integration of basic and value types of all the procedurally influenced languages, it is influenced by procedural languages, which becomes visible in the treatment of control structures and the presence of basic types. Although EIFFEL's solutions to language design have their own merits we regard them as violating the "no distraction" rule.

The "one paradigm suffices" approach of SMALLTALK should facilitate the adoption of the object-oriented paradigm for complete beginner's. The deviations from a traditional procedurally influenced style are said to be disadvantageous for students with some background knowledge, but can be regarded as a welcome leveling factor, i.e., equalizing knowledge advantages between students with varying backgrounds.

2.2 Saving difficult issues for later

The first language should enable students to engage in interesting (mini-) projects, without demanding them to master involved problems.

2.2.1 Garbage Collection

Languages without garbage collection are inappropriate, as manual memory management is too complex and error prone. Simply ignoring the fact that memory has to be reclaimed in the absence of garbage collection would foster a wrong attitude for the particular language. It would also deprive students of the feeling of "mastering the situation" and of "doing everything right so far" with regard to the current requirements.

2.2.2 Pointers

By the same token, an introductory language should not expose students to raw pointers. Object references and dynamic binding should be treated as natural and basic language concepts, without forcing primitive memory access mechanisms on students. Even better still, a language should not require a full appreciation of the difference between value variables and pointer variables.

2.2.3 Hybrid Languages

In general, all hybrid languages, for instance, supporting both procedural and object-oriented style are less appropriate. Although the support of multiple programming paradigms can be advantageous for mature programmers, there is always a potential of confusion for the beginner. Beginners do not know which of the several ways possible to achieve something to choose and can not distinguish problems of this kind from general, "normal" learning difficulties.

2.3 Constructivist Learning

Proficiency with a programming languages can not be taught it must be learned. As with cooking or other skills involving the need to have a master plan as well as being able to master all the details involved, it is best to practice as much as possible. Accordingly, education should be a mix of objectivist design (lectures) and constructivist design, i.e., project based tutorials.

The constructivist theory puts an emphasis on the learner rather than the instructor, aiming to achieve a shift from teaching to learning [13]. For the duration of a project a learner should be granted autonomy for self-directed learning. By hypothesizing, experimenting, and discovering learners can then actively construct their own understanding of subject matters.

2.3.1 Minimalism

The Minimalist theory of J.M. Carroll is a framework for the design of instruction [4]. According to the theory

- learning tasks should be meaningful and self-contained activities.
- activities should provide for error recognition and recovery.
- passive training should be minimized by allowing learners to fill in gaps themselves.
- realistic projects should be tackled as quickly as possible.

Building upon previous learner's experiences, activities are to be solved in a learner-directed fashion. Errors are regarded as learning opportunities.

2.3.2 Learning by Doing

> *" There really is no learning without doing."*
> – Shank and Cleary [33]

In order to encourage self-directed learning one does not tell students what to learn but provides them with learning opportunities [1].

Learning by doing has a good effect on how long lasting learning effects are, because newly acquired knowledge can be put in the context of self-made experiences.

In accordance to the minimalist school of thought project based learning should, therefore, provide

- a task,
- all necessary resources, and
- tools including those for error recovery.

Students should be encouraged to use resources of their learning environment and ask instructors in case they are stuck.

2.3.3 The Importance of Debugging

The above mentioned learning strategies foster experimentalism and encourage exploration. But what if things go wrong? Errors are inevitable and, in fact, negative results are just as important for learning as positive results. What should be avoided, however, is causing students to get despondent or agitated by errors overwhelming them [24].

Beginner's fate: Action paralysis

When faced with non trivial errors — often causing the program to crash — students are left without solving strategies. They helplessly wait for a tutor, who shall investigate the problem.

Certainly, powerlessness and the passiveness while awaiting the tutor's attention is a frustrating experience for students.

Tutors usually have an advantage due to their experience with common errors. The most important advantage, however, has to be the knowledge of how to use a debugger. Tracking down why a program produces the wrong result or what causes it to crash is much easier if a debugger is available and one knows how to use it.

It seems only logical that learning environments should include a good tool for debugging and that students should be taught debugging skills as early as possible. Actually, there is no reason why students should not be able to trace a piece of software before they are able to write one themselves! Thoroughly understanding how selected examples work, is a good preparation for the construction of self made solutions.

Students equipped with debugging skills require less supervision. In addition, in a study about introductory programming it was found that students whose teachers provided exemplary models and debugging techniques, scored higher in subsequent tests [21].

2.3.4 Supporting Learning Environment

Constructivist environments do not predetermine a sequence of activities. Rather, they seek to provide a supportive learning infrastructure.

> *"The programming environment provided to help the program developer use the language seems to be at least as important as the language itself, and probably more so..."*
>
> – RogerBailey [24]

Requirements for an object-oriented programming environment for software construction, thus, are:

- Access to classes in libraries, including source code.
- Availability of documentation for both language and libraries.
- Tools for experimenting with code snippets, inspecting results, and debugging software.

Development environments which look and feel the same as those used for procedural languages are, hence, inappropriate.

Even EIFFEL's "ebench" environment carries some procedural legacies with it, although it is much more inspired by object-oriented principles than typical environments for JAVA or other languages.

One unfortunate assumption is that there is a single entry point program that is compiled and run after all its parts have been provided [16]. Even though this entry point can be configured, the correct view would be that of a collaborating network of objects where an entry point loses its meaning. EIFFEL's "ebench" has no facility to interactively test little pieces of code without placing them into test programs.

The object-oriented paradigm is truly reflected in an interactive environment if it is possible to

- evaluate pieces of code on the fly.
- set up object networks without going through the effort to write main programs.
- make changes to code and immediately see the consequences.
- close a running session and return to the same object network later on.

Such an interactive environment can considerably reduce the complexity of getting the whole system right the first time. Environments that provide immediate feedback and encourage incremental testing also increase the user's understanding of the software [31].

3 Smalltalk Supported Learning

The single term SMALLTALK refers to a triad of

1. an object-oriented language,
2. a comprehensive class library, and
3. a development environment (see Fig. 1).

3.1 Object-Oriented Language

SMALLTALK was developed with child psychology in mind [14] and was actually used by children during its development. Programming with SMALLTALK was supposed to be a natural extension of thinking. For instance, the use of keyword parameters instead of a meaningless parameter list in parentheses testifies this intention. Assuming a hypothetical font creation interface, compare the common style of parameter passing, for instance, JAVA

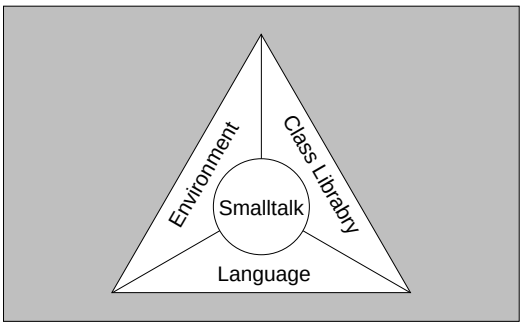

Figure 1: An integrated approach to interactivity

```
f = Font.create("Arial", 16, 8, false, true);
```

to the SMALLTALK version:

```
f := Font face: 'Arial'  size: 16@8
          bold: false  italic: true.
```

Not only do the keywords clarify the meaning of the arguments but also the font size can be passed as a single argument due to the easy creation of an appropriate point object. Recent environments for JAVA or C^{++} help programmers by popping up explanatory bubbles to flag the meaning of parameter positions. This kind of assistance is not available, however, when someone simply reads the code in order to understand it.

Beginner's fate: Difficulties with traditional syntax. —

A common JAVA error is to forget the empty parentheses "()" in the case of argumentless methods. Also, the ubiquitous semicolon is often forgotten or put at the wrong places after method definitions or loop headers. -

In the case of semicolons after loops or "if" statements the mistake goes undetected by compilers causing a lot of confusion due to the unexpected behavior of not looping the body or always executing the "if" branch. Also, when concepts like method declarations are still fuzzy it is not easy to judge whether a closing bracket is followed by a semicolon or not. SMALLTALK browsers provide templates for class definitions and offer single methods as the entity for code editing. Thus, the need for declaration syntax (e.g., to separate methods) is almost completely avoided. The absence of builtin control structures also minimizes the danger of placing separator symbols at wrong or dangerous places.

SMALLTALK, consequently, surely qualifies as a language that offers syntactic simplicity with a few special cases to remember [7]. Students have understandably little comprehension as to why the syntax for accessing arrays in JAVA is different than accessing other collection types. EIFFEL and SMALLTALK allow a uniform syntax for both cases.

Automatic garbage collection and the dominating theme to define everything with objects and sending messages, complete SMALLTALK's picture of a language for students causing minimal distraction when trying to convey programming concepts (see Sect. 2.1).

3.2 Comprehensive Library

A *standardized* library of base classes serves at least three purposes:

1. There is no need to reinvent the wheel time and again for standard data structures or deal with a variety of different competing libraries.

2. Students can write useful software and attack motivating problems from a very early stage on.

3. Browsing the source code of good libraries can reveal many tricks of the trade including programming conventions, language idioms, and design patterns.

Although, the following example is trivial, and makes minimal use of library functionality —

```
((1000 factorial) asString) size.
```

— it, first, demonstrates that the SMALLTALK library makes it easy to calculate an astronomical large number and then to determine how many digits it has and, second, points out how difficult the same task would be in a different language, say JAVA.

While the above line can be simply typed in and evaluated in a SMALLTALK workspace window, JAVA requires a full `main` program. Now the code excerpt using JAVA's `Math` class

```
l = Math.factorial(1000).toString().length();
```

raises a few issues:

- there is no `factorial` method in class `Math`.
- it uses a mixed procedural/object-oriented style.
- it requires funny empty parentheses.

If students were asked to provide an implementation for `factorial` their solution using JAVA integers would produce an (unnoticed!) overflow during the calculation of the factorial and, thus, a wrong result. Clearly, the rich SMALLTALK library with its sophisticated handling of large numbers represents a big advantage here.

In other areas as well SMALLTALK's library is easy to use, yet powerful. For instance, there is no need to introduce event handling before students can begin to use graphics. It is just one line of code to get a graphics window and another two to move and turn a graphics cursor:

```
Window turtleWindow: 'Tom the turtle'.
Turtle go: 50.
Turtle turn: 90.
```

The availability of a large number of collection classes allows instructors to adopt an approach in which students uses collection classes to complete interesting projects before they know how to implement such classes themselves. The study of the efficient implementation of data structures is then left to later semesters as opposed to performing unexciting and inefficient attempts at data structures before a higher level project can be tackled.

The comprehensive SMALLTALK library and the ease of its access within the integrated environment helps to foster two ideas in students:

- There is no main program. Programming means to pick the right classes and to occasionally create new ones.
- More often than not one has to (re)use rather than create, i.e., understand how to use already available classes.

3.3 Interactive Environment

The SMALLTALK class browser allows viewing and searching in a hierarchy of classes. Methods are presented for inspection and change only one at a time. This creates a very local focus of change and is quite different to a view of a single program with some internal borders called classes each of which stored in a file.

Even when student software (e.g., a video rental system) is running a change of code within a method can be put into immediate effect by just committing the change. If a method is changed only than is it automatically recompiled and used in the running system. If changes are made to an object's structure, these must obviously be deferred until no more of these instances are alive. In traditional compiled languages, beginners are typically confused by the need to recompile after making a simple change. Also, the need for their program to be complete before it can be run is not obvious to them [6].

As SMALLTALK does not impose static correctness for student software, it will run student code, even if it is in an incomplete state, until — if at all — code holes are encountered. This enables an evolutionary approach to software development where methods are defined as they are required, for instance, by refinement of an overall design. This helps to reduce a big complex task into stepwise small solutions and thus meets the requirement of a supportive environment (see Sect. 2.3.4).

The absence of a lengthy edit-compile-run cycle makes programming a much more interactive activity. Even the smallest programs in traditional languages make considerable use of standard libraries (e.g., input/output) which take a long time to check or precompile. In SMALLTALK the change of a method is as quick as opening up a workspace to try and test a few lines of SMALLTALK code. There is no need to create dedicated test programs in order to verify assumptions and make observations. Such conditions are clearly better suited to maintain student curiosity and support an explorative mode of study (see Sect. 2.3.2).

3.3.1 Builtin Debugging Facilities

An error of some fame is the inevitable "null pointer exception", "access violation", or "feature applied to void reference" error depending on the language used. The error occurs when a message is sent using an object reference that has not been initialized or is set to the null pointer for a different reason. When this happens in C++, JAVA, or EIFFEL more or less information is given as to where the error occurred, i.e., a trace of the execution stack is printed. The program is aborted and the student is then left to set breakpoints appropriately in order to rerun the program in debug mode.

In SMALLTALK a "walkback" window appears, explaining the problem. With a click of a button a debugger is available. It allows all methods in the calling history to be visited and in each context attribute values can be inspected. No time is wasted in finding the correct spot for breakpoints and to (often multiple times) step through loops until the right moment was tracked down. Typically, the breakpoint will be at a place where it is encountered several times before the conditions for the real error are really met. Instead of recreating the error in debugging mode, the SMALLTALK approach simply enables direct inspection after the the fact.

The debugger will accept any expression to be evaluated in the current execution context at request. Its control and inspection facilities make it rarely necessary to insert special test code into programs. The latter being a tedious strategy which nevertheless is often adopted by students using less advanced environments or with insufficient training in using a debugger.

Student empowering tools such as the readily available debugger, are a prerequisite for a successful self-directed learning mode of study (see Sect. 2.3.3).

3.3.2 Persistence Support

SMALLTALK's concept of a system image enables a snapshot of the system to be captured and continued with later. This includes all system and user objects, i.e., after setting up an example scenario students can simply preserve it by saving the image. There is no need to implement file handling first to store, for instance, video rental data. Again, this is useful to address interesting projects early without forcing too many implementation details on to students (see Sect. 2.3.1).

3.4 Adaptable Environment

Unlike other languages, SMALLTALK does not make a big difference between the (usually privileged) designer and the (usually less powerful) user of the language. Likewise there is no difference between a systems programmer and the ordinary software programmer. Figure 2 depicts how programming in SMALLTALK works. The class library is programmed on top of a virtual machine, making it portable. The development environment simply consists of additional classes, making use of the class library. Applications are written the same way. By simply adding further classes one can either change an existing application or adapt the development environment as required.

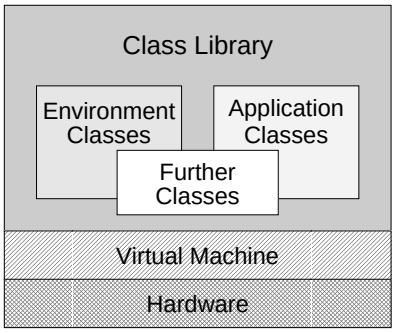

Figure 2: Modifying the system by adding classes

Instructors may choose to offer different browsers or message reporters. Furthermore, SMALLTALK already features different kinds of inspectors depending on the object type under examination and instructors may want to add to the list of specialized inspectors. These can display data in a tailored fashion and also offer a means to further follow the values contained in an object. Instructors, hence, may even enhance the suitability of SMALLTALK as a constructivist learning environment (see Sect. 2.3). There is no reason to accept any environment shortcomings since it can be moulded to accommodate any need.

3.5 VisualExpress

In our teaching experience, students have difficulty to relate static program text to the dynamic structures it creates and manipulates. However, they respond well to object diagrams, showing the existing objects with their containment relationships [18]. We have used such diagrams for set exercises, referring to model solutions.

In addition, however, we feel that the diagrams should even better reflect the student's individual solution at any particular stage. Hence, we started the VisualExpress project, which supports the automatic creation of object diagrams and single step tracing of student software under Smalltalk Express.

3.5.1 Why Smalltalk Express?

This particular SMALLTALK product easily fits on two floppy discs including an "HTML" online tutorial and a class encyclopedia. Unlike other platform independent SMALLTALK versions it is tailored to the ubiquitous Windows platform, the predominate operating system at student machines. Smalltalk Express is available free of charge [28] and runs very fast even on older machines with INTEL 486 processors. It comes with the complete software and documentation of a graphical user interface designer tool, which makes it useful for later, more advanced projects. Smalltalk Express browsers do not feature the usual SMALLTALK categories and protocols, which are very useful for organizing large software systems. In an educational context, however, these add complexity to the system and could be confusing to beginners.

There is an excellent book for SMALLTALK beginners which refers to Smalltalk Express. The book adopts a discovery approach to learning, i.e., takes the reader through developments rather than presenting heaven sent solutions [19]. The product, its documentation, and the book, therefore, complete each other to provide a constructivist learning environment (see Sect. 2.3).

3.5.2 Visual Perception and Tangible Control

When young children explore the world they use their (mostly visual) perception and act by using tangible controls. Likewise, for an intuitive programming environment it should hold true that

- relevant items should be visualized (e.g., an object diagram).
- the visual presentations should
  - allow closer inspection if desired, and
  - provide a means of manipulating the visualized elements.

We may summarize these principles as offering an object-oriented programming environment supporting direct manipulation. Different views (e.g., object diagram or method code browser) should accommodate the required viewing context and should also be easily accessible from each other.

However, just as we do not expect young children to use complicated tools — such as microscopes with the need too adjust focus and other parameters — if all they want is to take a closer look at an object, we should not ask students to deal with complicated mechanisms if all we want is to provide them with additional views. The provision of a direct manipulation interface should be completely transparent to students and their code. In

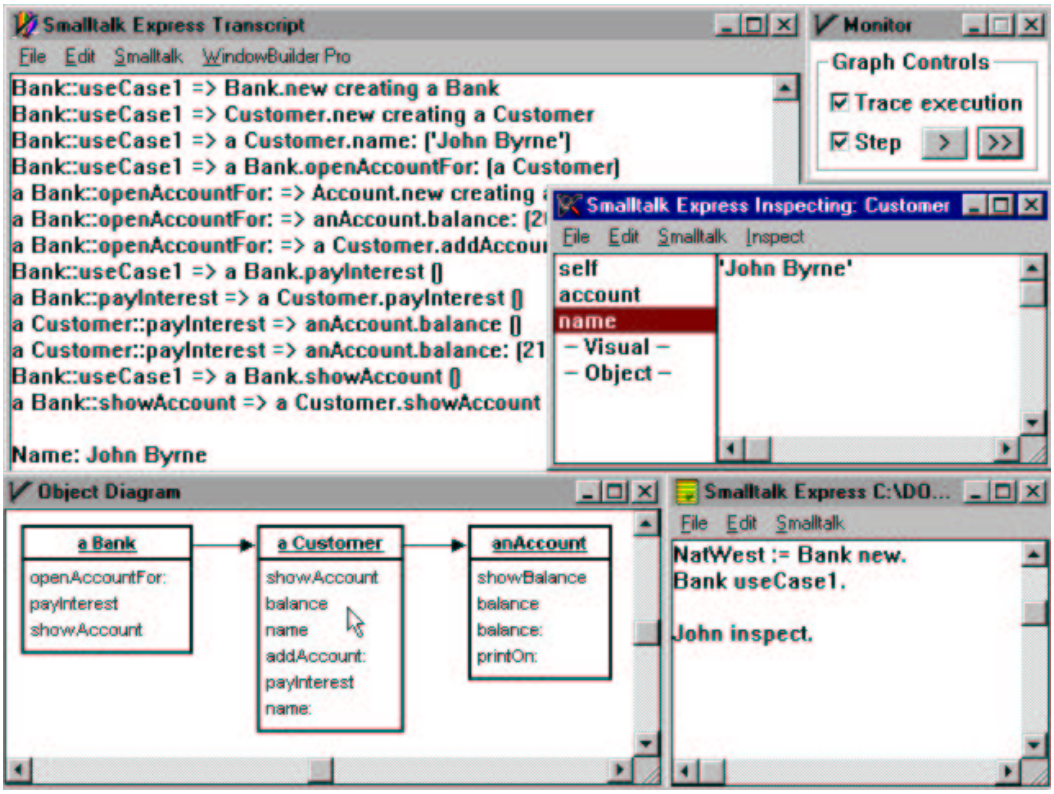

Figure 3: VisualExpress

particular, students should not be asked to insert additional lines of code which may be required to produce any of the views.

3.5.3 Why Object Diagrams?

Programming novices often literally do not know where argument values come from and where return values go to [18].

Beginner's fate: Lack of context awareness

Although methods offer all necessary data via parameters, students still try to get the data by explicit user console input. Results are often printed instead of being returned as method return values. Sometimes variables are assumed to be globally accessible as if there were no object boundaries.

As a consequence of this problem, some students wonder for instance, why it is necessary to assign constructor arguments to object attributes. Furthermore, the binding of argument values to method parameter is a new and alien concept for most students. Often they assume that the variable names have to be identical for this to work.

All these are symptoms of one problem: The big picture is missing. The latter can easily be provided by presenting object diagrams which depict how objects are connected to each other. By stepping through the program while updating the diagram, (e.g., object attribute values) a lot of the above misconceptions can be clarified.

Object diagrams should reflect the individual student code, but can be hard to construct for students by themselves. This is why VisualExpress automatically creates an object diagram during software execution. If desired, the execution can be done in a stepwise fashion allowing each method call to be followed and causing the diagram to build up gradually.

3.5.4 Diagrams for Free!

Figure 3 shows a screen shot from a sample Visual Express session. The lower right window is a workspace where a student started up an experimental bank scenario with the first two lines of code. The top right window shows controls that were used to trace the execution in single step mode. The big window at the top left shows each method call with source, target, message, and parameters. The object diagram at the lower left is the graphical display of the resulting object structure. A double click with the mouse on the customer object invoked an inspector window at the middle right for the customer object. The last line in the student code win-

dow shows an alternative way of achieving the same goal. The inspector not only shows object details but can also be used to alter them respectively.

In accordance with the remarks made in the previous section, no additional student code is required for all this to work. The sole action students are required to take is to derive their classes from class `Visual` instead of class `Object`. No further conventions need be obeyed. As a welcome side effect deriving from a single class other than `Object` causes all student classes to be available at a single spot through the browser. Normally, the browser would otherwise sort all student classes alphabetically into the large list of already available library classes.

The graphical object diagram naturally complements the code view on the software. Both code or even class diagrams present a static picture of a system. The object diagram, in contrast, shows the dynamic picture of software execution, i.e., it rolls out time into space, and can be exploited for animation effects (e.g., gradually building up the diagram or showing data travel between objects).

Activating single stepping through the VisualExpress monitor control has an effect on student classes only. Any use of library classes, such as collections, will be performed as atomic actions. This nicely confines the focus of investigation to a student class universe, preventing them to get lost in system implementations.

The VisualExpress prototype could be regarded as a visual debugger. For instance, it could be easily extended to even support switching single stepping on and off for each student class individually. Yet, the main purpose of this prototype is to provide the visualization of the object network for free.

The resulting diagrams may aid group work by providing a map to point to and talk about. They also tie the modeling view of objects (e.g., CRC cards) with the object-oriented model of execution.

3.5.5 Visual ADTs

Instructors may want to provide special graphical representations for certain classes, such as queues and stacks. They could be visualized in the same way as they are pictured in textbooks or the classroom, dynamically changing the display of their visual representation whenever, for instance, an element is added.

Instructors could either alter existing SMALLTALK data structures (see Sect. 3.4) or ask students to use specially provided ones. Again, without further display or animation code required by students, the object could be displayed either stand alone or as part of the object diagrams.

4 Where Other Languages (Are Thought To) Do Better

In the following sections we discuss several issues which have been used to argue against SMALLTALK.

4.1 Where pureness hurts

The "all objects and messages" approach of SMALLTALK has a slightly unfortunate effect on the way arithmetic expressions are interpreted. The expression

```
1 + 2 * 3
```

is evaluated to 9 as no precedence rules are applied but just message sending is interpreted from left to right.

On the other hand, SMALLTALK is one of the most faithful languages with respect to faithful arithmetic. Consider multiplying the number 1 ten times with the number $\frac{2}{3}$. Then multiply the result ten times with $\frac{3}{2}$. In C^{++} or JAVA the trouble starts with the expression `2/3` which is evaluated to zero because of integer arithmetics. EIFFEL fares better but calculates a result which, like the former languages when helped by using `2.0/3`, is close to but not exactly 1. SMALLTALK, on the other hand, automatically uses fractions and computes the exact result. Similar observations hold true for very large numbers which cause most languages to produce incorrect results. SMALLTALK will convert number types into each other (e.g., infinitely sized integers) whenever necessary or forced by the user. According to an anecdote, financial APL and SMALLTALK systems were the only ones which allowed their banks to continue making business while others were left without any means of action in a stock exchange crash.

The flip side of treating traditionally basic types as objects is that any initialization errors cause "message not understood errors". If a JAVA student forgets to set a variable `sum` to zero, the code will still work using the default initialization value for the respective type. In SMALLTALK any arithmetic operations will be sent to "`nil`" causing the mentioned error to happen. At least forgotten initialization is discovered this way but much better are default values in combination with compiler warnings.

Normally the SMALLTALK keywords syntax is very natural and parentheses are optional as in

```
(myPane pen) color: ClrBlue.
```

Sometimes, however, parentheses are required as in

```
(GraphPane openWindow: 'Fresh air') pen.
```

Without the parentheses message `pen` is sent to the title string of the window. Similarly, the expression

```
window backColor: ClrBlue foreColor: ClrRed.
```

yields "no such selector" although message `backColor` yields the modified window object. In this case, SMALLTALK reads one message with two keyword parameters instead of two messages with one parameter each. The correct expression is easily obtained by using parentheses or a ";" after the first message. This will cause the second message to be sent to the same window object.

Although, there seem to be some stepping stones there are, nevertheless, only four simple stages of evaluation order which can be easily memorized and applied. In general, the positive effects of SMALLTALK's syntax outweighs the few particularities. For instance, compare the SMALLTALK code to draw a rectangle with a turtle

```
4 timesRepeat: [
  turtle go: 50;
         turn: 90;
         flash
].
```

to the equivalent JAVA code:

```
for (int i=0; i<4; i++) {
  turtle.go(50);
  turtle.turn(90);
  turtle.flash();
}
```

While there are more similarities than differences, the SMALLTALK version has a syntactically simpler looping mechanism and enables multiple messages being sent to one object in a single expression.

4.2 Efficiency

Clearly, there are languages with a more efficient runtime model than SMALLTALK. Yet, for an educational context raw execution speed is not a consideration and has not prevented JAVA, which relies on interpreted byte code as well, to be widely used as a first language. For instance, both SMALLTALK and JAVA can be used to produce quick interactive animations as motivating examples for students.

4.3 Resemblance to Languages Used in Industry

Although there should be no doubt that a university's role is to teach concepts as clearly as possible rather than training students for an industrial setting, it can be argued that SMALLTALK's syntax has little resemblance to any of the main stream languages used in industry. The languages C, C++, EIFFEL, JAVA, PASCAL, etc. all share a similar ALGOL-like procedural syntax with builtin control structures. On the one hand, with any of these languages as a background it becomes easier to learn any of the other. On the other hand, there is quite a substantial industrial demand for SMALLTALK programmers as well, especially in the United States. Moreover, SMALLTALK's way of treating control structures conforms to object-oriented concepts rather than inheriting procedural solutions from earlier languages. This can help to teach objects right from the start as opposed to exercising loops and arrays for the main part of the course and using some objects at the end, time permitting. Also, as students learn to use blocks (snippets of code passed as parameters) right from the start, they will have no difficulty to understand and appreciate advanced designs involving patterns like Command [9] or Function Object [17].

4.4 Visibility Levels

In SMALLTALK, visibility rules are simple. Attributes are protected and methods are public. Methods are often marked as private with comments but there is no tool support. Classes always see everything of their superclasses, i.e., in C++ terminology there is no `private` modifier.

We consider it an advantage to teach the basic, object-oriented rules of attribute encapsulation without distracting keywords such as `public`, `protected`, and `private` or even an implicit modifier `package` as in JAVA. Optimization strategies and software engineering in the large as supported by EIFFEL's selective client export can be left for later semesters. Nothing wrong needs to be unlearned, anyway, as one could argue in the case of JAVA which rather doubtfully gives access to attributes by default.

4.5 Genericity

Genericity (also known as parametric polymorphism) is required in statically typed languages only. EIFFEL has generic classes, C++ offers templates, and JAVA rather unfortunately still has no support. With a dynamically typed language such as SMALLTALK there is no case for a genericity mechanism. Collections, for instance, work for all object types right away. There is no need to instantiate a class template for a certain element type. The loss, however, is that it is not possible to enforce monomorphic collections, for instance, ensure that only persons and no toasters wait in a queue. JAVA and SMALLTALK share the same problem when encountering a toaster: A runtime error would occur. In JAVA a "class cast exception" occurs in any event and in SMALLTALK a "message not understood" would be issued in case the toaster does not understand the message meant for a person. Of course, it is no problem to first dynamically check the element type and omit sending message if not appropriate. At least, no explicit casting as with JAVA is required for

code to work that would never create problems anyway.

In general, we believe that it is not a real problem in practice to avoid wrong object types being put into collections, especially considering the typical size of student projects. EIFFEL, with its clean albeit not perfect type system, again offers itself as a follow up language to investigate how monomorphic collections can be statically enforced in later semesters.

4.6 Static Typing

A static type system is regarded as an indispensable feature of a modern well designed programming language. While we agree in principle, the following sections shed some light on dynamic and static type systems in an educational context and the need to get static type systems as modern and well designed as the programming language aims to be.

4.6.1 Dynamic versus Static Typing

Although SMALLTALK has no static type system it, of course, has a dynamic type system. The only difference is that SMALLTALK

1. does not require students to annotate the software with types and

2. type errors are caught at runtime rather than compile time.

The benefits of additional documentation within the code due to type declarations is often emphasized as an advantage of static typing. We agree that it helps students clarify their thoughts and to understand their programs if they explicitely think about the object and variable types involved. Yet, we consider it an advantage to make type annotations optional in order to avoid typical beginner mistakes with static typing.

Beginner's fate: Confusion between declaration and use of variables

When passing parameter values students often prefix them with their type, just as the method declaration suggests. Variables are also often redeclared instead of simply being used, sometimes causing attributes to be shadowed by local variables.

Our teaching experience suggests that the notion of static types is an unfamiliar one to students and, worse, interferes with other difficulties such as learning the environment, syntax, algorithmic thinking and problem solving in general. Therefore, we consider it to be advantage of a first language not to impose static typing on students.

4.6.2 Other Ways to Record Type Information

Many functional languages facilitate programming by not insisting on explicit type annotations [11]. These languages still allow static type checking, something which is not possible with SMALLTALK. Notwithstanding, there are ways to document software with type information, using so called secondary notation.

Examples are the usage of comments as in

```
"dispose a vending machine item"
mc dispose: nextItem.
```

or an often applied convention to name variables according to their type. For instance,

```
library lend: aBook  to: aCustomer.
```

An even more helpful convention is to use a more elaborate naming scheme for keyword selectors as in

```
library lendBook: item  toCustomer: borrower.
```

It is true that secondary notation can not replace proper tool support. For instance, if a student forgets to put a return caret "^" in front of the value to be returned from a method, the method will return the receiver object instead. With a static type system the declared return type can be checked against the actual result type, saving the student the trouble to chase a chain of returns in order to find the error source. At least, Smalltalk Express prevents assignment to variables which refer to classes. The code

```
OrderedCollection := 2.
```

would otherwise be legal with dramatic consequences [12].

4.6.3 When Static Typing Becomes a Nuisance

Students sometimes reflect the desired static type of variables in their variable names but do not keep that consistent with the declared types. Consider the following JAVA example:

```
String num;

num = Console.readInt("Enter a number:");
```

Given the error message "incompatible types; found: int, required: java.lang.String" a first year student was unable to correct the problem. Although the method name `readInt` gives away its return type — which is not normally the case — and the `String` declaration is clearly visible, the student had to wait for assistance. Interestingly, had it not been for the erroneous `String` declaration the program would have run correctly, since all uses of `num` were referring to integer operations only.

We already hinted at typical problems with basic types in procedurally influenced programming languages (see Sect. 4.1). One of the most common beginner mistakes is to declare integer variables and subsequently try to calculate, for instance, an average. The result will simply not be correct due to integer division semantics. Clearly, students have to make these experiences in order to learn these issues in such languages. The question is "When?". When they struggle with other beginner problems and could do without further complications? Or later, when they have already mastered other initial hurdles (see Sect. 4.6.1).

A drastic example of how static typing can ruin all attempts to run an almost correct student project is the following JAVA bug:

Beginner's fate: Declaring a constructor with return type "void"

In an attempt to provide a correct type signature for a constructor method students declare its return type as `void`. This has the dramatic effect of making the constructor an ordinary method which will not be called upon object initialization.

This bug will cause very nasty runtime(!) errors including crashes. It is quite hard to find for instructors but next to impossible to correct for students.

4.6.4 Learning from Experience

With static typing students are forced to get the overall structure right before they can start experimenting with the system. On the one hand, this provides motivation to approach the construction of software with a clean and thought out plan, but on the other hand, it can be frustrating for beginners to satisfy all typing problems – urgent or not – before they can start to use and explore their solutions. Students forced to wait for an instructor before they can get a grip on the problem are unlikely to enjoy programming for the hours necessary to develop proficiency [5].

A type error reported early may save difficult debugging sessions required to find the source of the mismatching types. However, exactly such intense explorations may be instrumental in developing a desirable degree of program comprehension [22]. With static type errors the student still has to identify which of the two – or maybe both – type declarations need to be changed.

It seems much more helpful to let beginners *experience* their errors by running into them, rather than *warning* of potential problems by reporting type errors. In other aspects of education we also prefer the "see for yourself" approach over the "Not yet right" attitude.

Especially in the case of more complex type errors, the required program comprehension and the cognition as to why the error occurs, can be much better acquired if it is at least possible to step through the execution until the error occurs. In a controlled experiment it was found that reporting static type errors can improve productivity and remove a certain type of errors in final programs [32]. However, these advantages came only into effect after most other problems had been mastered. It was found that static typing had no effect on defect prevention (as opposed to defect removal). Most importantly, it was furthermore found that type checking is unlikely to help gain a better understanding.

Hence, we conclude that tools for production code in industry have different requirements, i.e., quick error removal in final programs, than environments in an educational context. In the latter case the more natural error removal cycle and bug tracing process fostered by dynamic typing is to be preferred since it enables an evolutionary software development with a deeper understanding of introduced bugs.

4.6.5 Understanding Error Messages

How do languages and environments behave if erroneous code is entered? In the following example a student sends message `init` to class `Apple` instead of to the freshly created apple object.

```
Apple new; init.
```

Will the student understand the JAVA message "Error (17) non-static method init cannot be referenced from a static context."? Or is the C^{++} message more helpful: "Improper use of typedef 'Apple'"? Or is it better to choose "debug" from the walkback window and experience where the message `init` is sent to? The latter investigation makes it easy to recognize the problem and remove the superfluous ";" in the code. At least in this example static typing does not present an advantage whatsoever, as the reported messages are next to useless for students.

4.6.6 How To Specify Conformance?

One advantage of static typing is that the type hierarchy serves as a conformance hierarchy. All subclasses of a class promise to behave in a compatible manner. Any accidental structural conformance between classes, i.e., when a relevant subset of the method signatures is identical but semantically large differences exist, is ruled out by tying the type hierarchy to polymorphic substitutability. With a statically typed language a programmer, thus, can ensure that only semantically conforming classes can be substituted for each other.

In fact, the difference to a dynamically typed language again boils down to checking at compile time or at runtime. It is a very simple matter of checking an object's

position in the SMALLTALK class hierarchy and to raise an exception if required.

Although it is nice that a developer can state the semantic intent of a class by putting it into certain place in the class hierarchy, there is no guarantee that it will actually conform. Even if a sophisticated assertion mechanism as in EIFFEL is available, this is more akin to cleverly integrated test code and leaves a lot to be desired compared to proofing semantic properties.

Needless to say, the requirement for a class to sit in a certain hierarchy is sometimes a hindrance for using perfectly suitable classes which happen to come from a different source.

In SMALLTALK the class hierarchy is used to realize subclassing, i.e., code reuse. As objects can freely be substituted for each other as long as they structurally conform this does not compromise the degree to which polymorphism can be exploited. Statically typed languages typically are forced to unify all subclass, subtyping, and "is-a" hierarchies, therefore, diminishing the power of each individual choice [20]. The language SATHER is an exception, featuring both a code and a type hierarchy respectively [27].

4.7 Programming by Contract

EIFFEL advocates a "programming by contract" style of software engineering which assigns responsibilities for client and server classes respectively. While clients make sure that the preconditions of their servers are met, servers in turn guarantee to fulfill their postconditions.

This idea is a very useful principle especially if supported by the programming language as in EIFFEL. We feel, however, that it is not crucial to *enforce* this software engineering principle from the very start. Students could be told how to reasonably distribute responsibilities but should not be expected to formulate pre- and postconditions to be included in the code.

Although, SMALLTALK traditionally does not feature language support for programming by contract, its reflection capabilities should make it possible to provide one. In any case, EIFFEL could again fill this gap in later semesters.

4.8 Multiple Inheritance

SMALLTALK is a single inheritance language, as is JAVA. JAVA's multiple interface inheritance does not gain any advantage over SMALLTALK, which accomplishes the same with unrestricted polymorphism.

Other languages do feature multiple inheritance, although it is not a generally accepted concept. It has been argued that more dynamic software is possible with object composition (see the BETA language [23] or a range of design patterns such as the bridge pattern [9]). In any event, multiple inheritance is not a crucial feature for a beginner's language. On the contrary, most languages have complex multiple inheritance semantics and, hence, are not suitable as a introductory language (EIFFEL being a noteworthy exception).

4.9 Exception Handling

Although there are no keywords in SMALLTALK to deal with exceptions, there is a sophisticated exception mechanism available. Again, by using an "all objects" approach and exploiting some reflective properties of the SMALLTALK system — such as access to the calling stack — the resulting system is simple but powerful and very elegant. It can be guaranteed that certain code will be executed whether exceptions occur or not. It is possible to specify exception sets and filters. Some exceptions can be resumed and all can be translated to other exceptions if desired.

SMALLTALK exceptions do not make themselves visible in student code. With JAVA as early as basic input is concerned one has to deal with explicit exceptions. These either have to be hidden with non-standard libraries or have to be explained at an unacceptably early stage.

5 A Beginner's Language Only?

In the preceding section we argued that a few concepts missing in SMALLTALK can be left for later semesters. This begs the question: Is there a life for SMALLTALK after the first semester? Luckily, SMALLTALK lends itself to use for a variety of advanced topics:

Operating Systems The reflective architecture as well as the exception mechanism could be subjects of investigation in operating system courses. Also, the multitasking capabilities and synchronization mechanisms are viable topics.

Software Engineering

- There is no better choice than SMALLTALK to illustrate the RAD (rapid application development) process.
- A so called refactory browser is available allowing students to restructure software which evolved into ill-structured designs.
- The meta modeling aspects in SMALLTALK are becoming increasingly important in the era of component based software development.

User Interfaces A freely available graphical user interface builder can be used to enable the easy production of fancy interfaces and to learn about the underlying event handling mechanisms.

Internet applications There are SMALLTALK web servers as well as SMALLTALK web clients and applets. Programming can be performed at any level, down to low-level socket handling.

6 Related Work

Kölling et al. analyze the requirements for a first year object-oriented teaching language [15]. Although they propose a new language, their ideal language/environment pair basically describes SMALLTALK. The only criticism they raise against SMALLTALK are the unusual syntax, the size of the class library and dynamic typing. We already discussed in Sect. 4.6 that in an educational context it is beneficial not to complain about errors until runtime to encourage experimentation and self-directed learning. Also, the fact that errors may not be discovered at all due to the lack of testing coverage is an advantage in our view. Programs by first year students do not need to be "cleanroom developments". A bug left sleeping alone in a dark corner is better than an annoying type error causing the student to seek help from an instructor.

The authors also claim that the big size of the library poses a problem because it has to be extensively used from the start. We do not understand why the use of an integer should be any different if it is part of the language or the library. For all intents and purposes students do not need to care about the difference.

It is true that, for instance, iteration is dealt with through the SMALLTALK library instead of with plain language features. But this is how it should be. Any instructor serious about teaching object-orientation should not set exercises asking students to write loops over arrays. Iteration is a matter of containers and should be offered as a service. The SMALLTALK notation required is minimal and the conceptual simplicity of its iteration approach compares favorably to the more complex dedicated iterator abstractions necessary in JAVA or EIFFEL.

Apart from the fact that learning to use existing code is a desirable thing, there are means to hide the richness of the SMALLTALK library from students. A class hierarchy browser can easily be opened on an arbitrary subset of classes. This way only relevant classes can be brought to a student's attention.

It is true that learning SMALLTALK involves learning a language, a library and an environment at the same time. If not approached properly this can be an overwhelming experience. One can, still, exploit the fact that the three parts facilitate the understanding of each other:

- The environment helps to understand the library.
- Library browsing helps understanding the language.
- Inspectors help understanding objects.
- The debugger helps understanding language semantics (e.g., method lookup).

Employing the "spiral pattern" [2] one can introduce a number of topics early without going into depth. This allows students to work on interesting problems, though they are not masters of any of the tools. Each topic is then repeatedly visited in order to achieve depth as well.

The last point of criticism from Kölling et al. concerns the unusual syntax. From a beginner's point there are no crucial differences between,

JAVA:	and	SMALLTALK:.

`if (x>y) {`
`  max = x;`
`}`
`else {`
`  max = y;`
`}`
		`x>y ifTrue:`
		`  [max := x]`
		`ifFalse:`
		`  [max := y]`

The SMALLTALK code could be considered more intuitive and object-oriented, especially if the examples are correctly coded using library features. In SMALLTALK —

```
biggest :=  x max: y.
```

— one uses the `max` method in class `Magnitude`. With JAVA —

```
biggest = Math.max(x,y);
```

— the code can not be object-oriented because `int` is a basic type and, thus, a procedural style must be chosen for the mathematical library.

Recent pedagogical proposals suggest a top-down curriculum as opposed to the traditional bottom-up approach. Both Meyer's inverted curriculum [26] and Callaghan's "Model and Implement" pattern [29] suggest the syntax-driven, bottom-up teaching track to be abandoned. The ability to directly start manipulating objects and reusing other people's code from day one in SMALLTALK makes it an ideal medium for this kind of curriculum. As a result, software can be understood as a tool for solving real world problems rather than an instruction set for a machine.

There have been a number of approaches to facilitate teaching SMALLTALK. The LearningWorks environment developed by Goldberg is used for the Open Universities course M206 entitled "Computing: An Object Oriented Approach". By their own account it is the world largest computing course with over 5000 students. LearningWorks builds on the VisualWorks SMALLTALK product and uses learning books to guide students from small to larger sections of the SMALLTALK world [8].

Another environment, "Rehearsal World" is based on a theater metaphor [10]. SMALLTALK objects are visualized as actors and programmers can invoke messages by

inserting code into cue sheets. While this seems to be an excellent metaphor to introduce object-oriented concepts and could well be used as a demonstration in lectures, we feel that it is best for students to work with the real language and environment from the start. According to the minimalist theory (see Sect. 2.3.1)) authentic tasks should be approached as early as possible [13]. Students should feel that they accomplished something of value in the real world. It is certainly motivating to consider that an employer would have paid them money to do a similar task.

Moreover, proficiency with a language also includes knowledge of its libraries and tools [30]. To "know your way around"means knowing where to find things, how to react to certain problems, etc. Hence, proficiency with a language can not as readily be acquired by learning in "sandboxed" environments.

7 Conclusions

We have demonstrated that the language, library, and environment triad called SMALLTALK can be used as a student enabling environment. Even better, it can be easily enhanced to improve its support for self-directed learning.

We presented the VisualExpress project as an example of how to improve exploration facilities without complicating the environment or student code. Useful dynamic visual diagrams are created without requiring students to write a single extra line of code. The browser's local code view on single methods and the visual, animated object map created by VisualExpress object diagram complement each other to provide a big picture where detail is readily accessible. The VisualExpress features, including single stepping student code and special instructor provided data structure visualization classes are completely non-intrusive in terms of student code. As a result, students get educational support but still use a real environment to perform authentic tasks. If such a student enabling approach also helps to answer the global phenomenon of having more and more students to teach with less and less resources then all the better. But primarily, constructivist learning should be adopted due to its higher learning potential for students.

We identified a number of issues that students can be spared with when choosing SMALLTALK as the first language. Manual memory management, pointers, integer overflow, arithmetic precision problems, etc. are, of course, worthwhile studying. The point we are trying to make is that there is room for all this in lessons two, three, and four. Lesson one, though, should provide a gentle introduction in order to avoid despondency or agitation at early stages.

We also investigated a number of alleged disadvantages of SMALLTALK. The quintessence of Sect. 4 is that some points of criticism can be regarded as advantages with a constructivist stance to learning. Furthermore, those features that are really missing from SMALLTALK have been identified as non-essential for a first contact with object-orientation.

The Smalltalk Express product is available free of charge to students and instructors. In combination with the accompanying documentation, tools, and possible additions such as VisualExpress it constitutes a constructivist learning environment where learners use a variety of tools and information resources in their aim to achieve a goal.

We found it refreshing to discover how a seemingly old language and environment provides perfect answers to modern educational ideas.

8 Acknowledgments

The author would like to thank Colin Atkinson for his helpful comments.

References

[1] H. Barrows. *The tutorial process.* Southern Illinois University, 1988. 2

[2] J. Bergin. Spiral pattern. In *The Pedagogical Patterns Project: Successes in Teaching Object Technology.* Project leaders: http://www-lifia.info.unlp.edu.ar/ppp/, July 1999. 13

[3] D. Boud. Researching learner-managed learning: Seminar response part II, http://www.lle.mdx.ac.uk/iclml/response.html. In *The International Centre for Learner Managed Learning.* Middlesex University, October 1998. 1

[4] J.M. Carroll. *The Nurnberg Funnel.* MIT Press, Cambridge, Massachusetts, 1990. 2

[5] S. M. Carver. Learning and transfer of debugging skills: Applying task analysis to curriculum design and assessment. In R. E. Mayer, editor, *Teaching and Learning Computer Programming: Multiple Research Perspectives*, pages 259–297. Lawrence Erlbaum Associates Inc, NJ, 1988. 11

[6] B. du Boulay. Some difficulties of learning to program. In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*, pages 283–299. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989. 5

[7] B. du Boulay, T. O'Shea, and J. Monk. The black box within the glass box: Presenting computing concepts to novices. In E. Soloway and J. C.

Spohrer, editors, *Studying the Novice Programmer*, pages 431–446. L. E. Assoc., Hillsdale, NJ, 1989. 4

[8] A. Goldberg et al. The learning works development and delivery framework. *Communications of the ACM*, pages 78–81, October 1997. 13

[9] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison-Wesley, 1994. 9, 12

[10] L. Gould and W. Finzer. Programming by rehearsal. Technical report, Xerox Palo Alto Research Center, May 1984. 13

[11] Hudak, Peyton-Jones, Wadler, Boutel, Fairbairn, Fasel, et al. Report on the programming language Haskell: A non-strict, purely functional language v1.2. *ACM SIGPLAN Notices*, 27(5):Section R, May 1992. 10

[12] R. E. Johnson. Classic Smalltalk Bugs. University of Illinois at Urbana-Champaign, http://st-www.cs.uiuc.edu/ftp/pub/Smalltalk/st-docs/classic-bugs, 1998. 10

[13] D. H. Jonassen. Objectivism vs. constructivism: Do we need a new philosophical paradigm? *Educational Technology: Research and Development*, 39(3), 1991. 2, 14

[14] A. Kay. The early history of Smalltalk. In T. J. Bergin and R. G. Gibson, editors, *History of Programming Languages 2*, pages 511–578. Addison-Wesley, 1996. 3

[15] M. Kölling, B. Koch, and J. Rosenberg. Requirements for a first year object-oriented teaching language. *SIGCSE Bulletin*, 27(1):173–177, March 1995. 13

[16] M. Kölling and J. Rosenberg. An object-oriented program development environment for the first programming course. In *Proceedings of the 27^{th} SIGCSE Technical Symposium on Computer Science Education*, pages 83–87, March 1996. 3

[17] T. Kühne. The function object pattern. *C++ Report*, 9(9):32–42, October 1997. 9

[18] T. Kühne. Teaching java for first years: Typical difficulties and common mistakes. Recorded personal experiences, Staffordshire University, 1998. 1, 6, 7

[19] W. LaLonde. *Discovering Smalltalk*. Benjamin / Cummings Publishing, 1994. 6

[20] W. LaLonde and J. Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991. 12

[21] M. C. Linn and J. Dalbey. Cognitive consequences of programming instruction. In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989. 3

[22] F. J. Lukey. Comprehending and debugging computer programs. In M. J. Coombs and J. L. Alty, editors, *Computing Skills and the User Interface*. Academic Press, 1981. 11

[23] O. L. Madsen, K. Nygaard, and B. Möller-Pedersen. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley and ACM Press, 1993. 12

[24] J. L. Martin. *Is Turing a better language for teaching programming than Pascal?* B.sc. computing science and education honours dissertation, University of Stirling, January 1996. 3

[25] B. Meyer. *EIFFEL the language*. Prentice Hall, Object-Oriented Series, 1992. 2

[26] B. Meyer. Toward an object-oriented curriculum. JOOP: Education & Training, May 1993. 13

[27] S. Murer, S. Omohundro, and C. Szypersky. Engineering a programming language: The type and class system of Sather. In Joerg Gutknecht, editor, *Programming Languages and System Architectures*, pages 208–227. Springer Verlag, Lecture Notes in Computer Science 782, November 1993. 12

[28] Smalltalk Express: A free Smalltalk/V and WindowBuilder Pro/V. Object Share, http://www.objectshare.com/products/smalltalk/se/seinfo.htm, 1999. 6

[29] A. O'Callaghan. Model and implement pattern. In *The Pedagogical Patterns Project: Successes in Teaching Object Technology*. Pedagocial patterns project leaders, http://www-lifia.info.unlp.edu.ar/ppp/, July 1999. 13

[30] D. N. Perkins. Preface: Minds in the hood. In B. G. Wilson, editor, *Constructivist learning environments: Case studies in instructional design*. Englewood Cliffs NJ, 1996. 14

[31] D. N. Perkins and F. Martin. Fragile knowledge and neglected strategies in novice programmers. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 213–229. Ablex Publishing Corporation, Washington DC, 1986. 3

[32] L. Prechelt and W. F.Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering*, 24(4):302–312, April 1998. 11

[33] R. Shank and C. Cleary. Top ten mistakes in education. In *Engines for Education*, pages 181–213. The Institute for the Learning Sciences, Lawrence Erlbaum Associates, 1994. 2