Introductory Computing: The Design Discipline

Viera Krňanová Proulx[*]

Northeastern University, 360 Huntington Ave.,
Boston, MA 02115, USA
vkp@ccs.neu.edu

Abstract. The goal of this paper is to present in context the key didactical principles behind the *Program by Design* curricula, motivate the need for the supporting software, and describe in detail the *How to Design Classes* component for teaching introductory object oriented program design using Java and Java-like languages. The key innovations are a systematic test-first program design, and the introduction of programming language concepts by designing abstractions based on existing programs.

Keywords: Informatics in primary and secondary education, design principles, software for novice programmers, abstractions.

1 Introduction

What is computing? What is informatics? The answer to this question guides the design of the curriculum that focuses on the principles, not fads. At the heart is the computation: a program that consumes data and produces new data according to some formula. But this is just basic algebra, automated. To extend this notion of computation, we need to deal with more complex data. No algorithm exists apart from data. What comes first? We believe that understanding data, how information can be represented as data, and how data conveys information is at the heart of computing and deserves a serious place early in the curriculum. Well-structured data reveals clearly numerous algorithms for extracting new information, and provides the context for learning the foundations of program design. The key questions: the design of abstractions, the concerns about efficiency, the multiple ways the same information can be represented as data, the difficulty of reliable and secure data transmission, the communications protocols, and many others arise naturally in this context.

The traditional curricula for introductory programming start by designing algorithms and overwhelming the student with complex syntax and language features, but providing little guidance on what the program design process should be. Out of more than 20 Java-based textbooks only two mention testing of students programs, and even then without the appropriate software support [1], [13]. Other recent approaches use game-like environments to make the programming more attractive and accessible [3], but still fail to focus on the design process that guides the program design. Tinkering, trial and error approach rules.

[*] Partial support for this project has been provided by the two NSF grants DUE-0618543 and DUE-0920182.

I. Kalaš and R.T. Mittermeir (Eds.): ISSEP 2011, LNCS 7013, pp. 177–188, 2011.
© Springer-Verlag Berlin Heidelberg 2011

In the first part of this paper we describe the key ideas of the *Program by Design* project that introduces systematic program design principles for students ranging from grades 6-7, all the way to the university level. The various components of the *Program by Design* curriculum have been developed over the years by a team of programming language researchers, software developers, and educators. The author has had a key role in the design and implementation of some of the libraries, and in the design and implementation of the most advanced component of this curriculum, formerly known as *ReachJava*.

In the second part we focus on the *ReachJava* component, that presents the design of programs in object-oriented Java-like languages. We describe the role of supporting libraries that expose to the novice programmer the essential design principles while hiding the confusing detail. In the third part we show how this approach leads naturally to a systematic design of abstractions and provides the context for understanding more complex programming language features, as well as the design and the use of standard libraries.

2 How to Design Programs

The foundation of *Program by Design* (known as *TeachScheme!*) has been presented in the textbook *How to Design Programs* [4, 5] and its German language counterpart *Die Macht der Abstraktion* [8]. During the past five years, the curriculum has been adopted and augmented, with dedicated software support, to target young children, ages 10-13. The *Bootstrap* curriculum has been taught to hundreds of children and all materials are available online [1]. The *How to Design Programs* curriculum is appropriate for all students in secondary schools and universities, regardless of their field of interest. While the context of these curricula is the design of programs, the ultimate goal is to teach the students fundamental skills for solving complex problems and organizing the solution in a systematic way.

Functions and Algebra: Bootstrap. Typical first programs students often encounter involve designing and evaluating a simple algebraic function: compute where will a cyclist be after the given time elapsed, if he is traveling at the speed of 25 km/h. We see that the distance is a function of time and can be written as *distance = fnc(time)*. We can explain this idea through simple tables: at times 0, 1, 2, 3, the cyclist will be 0, 25, 50, 75, km away. But with the right programming environment, we can turn these functions into controls of an interactive animation: the movement of the cyclist is represented as a function that for each tick of the clock produces the current location of the cyclist on the screen. Now, what if the response to the *left* and *right* arrow keys that moves another object horizontally is encoded as another function. We add the detection of a collision as a third function, and we have finished programming the model of an interactive game. This is the beginning of the *TeachScheme!* curriculum and is the key feature of the *Bootstrap* curriculum. Children in the Bootstrap program write down the list of locations where the falling ball will be after each tick of the clock, then design the functions that model the movement. The basket catching the ball at the bottom moves in response the keys pressed. The conditional (a function that produces a `boolean` value) is used to update the score. The image of a ball or of a basket is a simple primitive data item in the program. The drawing of the

images on the `canvas` (the game board), the invocation of the event handlers (the functions `on-tick` and `on-key` defined by children) and the entire animation is controlled by the provided library.

This is serious work. Children are true designers, learning basic algebra to implement their games. After nine lessons they can explain the evaluation of expressions, the substitution principle, the conditionals, and proudly show their game.

Taking Design Seriously. The didactical principles of the Program by Design curriculum are based on enabling the learner to master a systematic approach to problem solving by following a well-structured design process encoded in three *design recipes*. The *design recipes* give the instructor a tool to diagnose the student's problems by identifying the step in the design process in which the student encounters difficulties.

When we teach children to design functions, we give them a blueprint, a roadmap that shows them the steps in the design process. Once we have identified the data needed to represent both the inputs, and the expected result, we follow the ***design recipe for functions/methods****:*

- Write down in English the purpose statement for the function/method, describing what data it will consume, and what values will it produce. Add a contract that specifies the data types for all inputs and the output.
- Make examples of the use of the function/method with the expected outcomes.
- Make an inventory of all data, data parts, and functions/methods available to solve the problem.
- Now design the body of the function/method. If the problem is too complex, use a *wish list* for tasks to be deferred to helper functions.
- Run tests that evaluate your examples. Add more tests if needed.

The children's version is adapted to their abilities, but the focus on systematic design remains. The comment from a child *'I never knew I could divide a big problem into smaller ones'* affirms that these design principles transcend computing and programming. Seasoned programmers recognize that we practice ***test-first design***.

Understanding Data. After the first brief introduction to representing simple programs as functions (that correspond directly to mathematical functions) the *How to Design Programs* curriculum focuses on understanding the complexity of data, the way how information can be represented as data, and, conversely, how data can be interpreted as the information it represents.

The first step in designing a program is always the design of data that represents the problem. The ***design recipe for data definitions*** guides the students as follows:

- Can you represent the information by a primitive data type?
- Are there several related pieces of information that describe one item? If yes, design a composite data type (*struct, class*).
- Does the composite data type contain another complex piece of data? Define that data type separately and refer to it. (A `Book` data item contains an `Author` data item.)

- Are there several variants of the information that are represented differently, but are related (*e.g. a circle, a rectangle, a triangle --- all are shapes*)? If yes, design a union type. (In Java, define a common `interface`.)
- Repeat these steps. This may lead to self-reference, mutual reference, and eventually to a complex collection of classes and interfaces.
- Make examples of data for every data type you design.

Students learn to design complex data: ancestor trees (with person's mother, father, their ancestors); data that represents files and directories in a computer system; ice cream cones with the cone and a list of toppings; a river system with confluences and tributaries; etc. When designing functions for such complex data, the inventory step of the design recipe calls for identifying not only the function inputs, but also the parts of any composite data (*struct, class*), variants of a union type, as well as all functions that are already available for either the input data or the parts of the input. So, if one of the shapes is a combination of the *top* and the *bottom* shape, any function defined for shapes can be used for both the *top* and the *bottom* parts.

Simple language, complex data, serious program design is our motto. All of this can be taught in the context of a very simple language that supports only the appropriate data definitions (with their constructors, selectors, and predicates that identify the data type) and on the functional side provides the standard arithmetic, relational, and logical expressions, and a conditional. If every function produces a new value, the result, then the entire design process is very straightforward:

- Tests are simple, as they only verify that the result matches the expected value.
- Function composition comes naturally, result of any function application can be used in further computations.
- The order of computation does not affect the result. (However, a function or a data item must be defined before it can be used.)

To provide fun and challenge, we provide libraries that handle interactive graphics back ends of game, with students designing the model: the functions that produce a new scene in the game in response to a key event, or timer tick. Drawing scenes using shapes and images is supported through functions that support the composition of images. Games like *pong*, *snake*, *space invaders*, provide a design playground.

Designing Abstractions --- Advanced Programming. After we have written several programs (functions/methods) that solve similar problems we begin to see patterns: the solutions are very similar to each other. Students see that certain functions appear similar, the way the data is handled follows the same pattern, or that some code needs to be repeated. To simplify the code and to eliminate repetition, students see the need for more complex programming language features. Rather than using existing libraries to illustrate the generalized solutions, our goal is to teach students how the libraries are built. To achieve this goal, we present a systematic design process encapsulated in the ***design recipe for abstractions*** that helps us eliminate code repetition and produce a more general solution:

- Mark all places where the similar code segments differ.
- Replace them with parameters and rewrite the solution using them as arguments.

- Rewrite the original solutions to your problems by invoking the generalized solution with the appropriate arguments.
- Make sure that the tests for the original solution still pass.

The *How to Design Programs* curriculum now follows with the introduction of local variables, functions as function arguments, mutation of data, as well as the discussion of the efficiency of computation, and additional more advanced topics.

The three ***design recipes*** are at the heart of the *Program by Design* curriculum. They embody the core questions all programmers face and give the student a guide through the design process. They correspond to the three cornerstones of our curriculum: understanding the connection between information and data and the importance of the design of complex structured data, using the test-first design process for the design of every function or method, and understanding the process of abstraction that turns a problem-specific solution into a generalized solution applicable to a collection of related problems.

3 How to Design Classes

The *Program by Design* curriculum has as its goal to provide a systematic introduction to the fundamentals of computing and programming. The ideas introduced at the beginning apply equally well in a more complex context. The *ReachJava* component with the draft of a textbook *How to Design Classes* [4] extends the original *TeachScheme!* curriculum to the context of class based programming using Java-like languages by introducing most of the essential concepts of object-oriented program design. It is appropriate for secondary schools and universities.

The goal of this section is to reflect on what we learned during the last nine years of implementing the *ReachJava* curriculum and designing the supporting software. We start by showing how the pedagogical principles of *Program by Design* imply the need for novice-appropriate software libraries that support this methodology. We then show how the *ReachJava* curriculum teaches students through systematic design of abstractions to build reusable software and to use standard software libraries.

Libraries for Novice Programmers: FunJava. While many functional languages (such as Scheme) have a compact and fairly simple syntax (at least at the beginner's level), statically typed object-oriented languages such as Java or C# require a complex syntax for solving even the simplest problems. To eliminate a number of problems novices face, our curriculum starts with a limited version of Java (***FunJava***). There is no assignment statement, all fields are initialized either when defined, or in the only constructor allowed. A `class` can implement only one `interface`, and there are only two statements: `if` with a required `else` clause, and `return expression`. This enforces a mutation-free programming style, the original goal of the designers of object-oriented languages. Every method produces a new value, a new instance of data. Rather than starting with *algorithms*, we first practice designing classes and collections of classes and interfaces that represent different, gradually more complex, information. Students design classes that contain fields that are instances of another class, unions of classes, self-referential data, mutually-referential data. The earlier

examples of data: ancestor trees, a model of a river system with a number of confluences, the representation of computer files and directories (that contain other files and directories), the representation of a route through the cities, a student's record with the list of courses she is enrolled in, now define a collection of interconnected classes and interfaces.

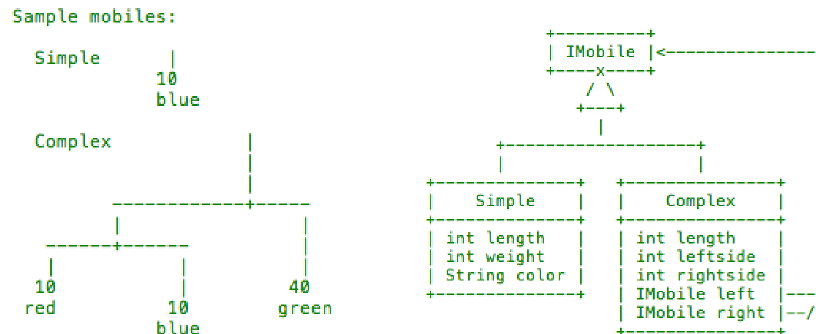

Fig. 1. The examples and a class diagram for a program that models a hanging mobile

Libraries for Novice Programmers: Tester Library. The *design recipe for data definitions* guides the design decisions and teaches a systematic approach to understanding the complexity of data. We use a simple version of class diagrams to illustrate the relationships between classes and interfaces: the containment and the inheritance.

Once we have examples of classes and data, we turn to designing methods, following the same *design recipe*. Functions become methods, and the object that invokes the method (`this`) becomes just an additional argument the method consumes. Without mutation, the outcome of every method depends only on its inputs. So the students only need to check that the outcome of a method invocation produces the desired value.

```
// is this mobile balanced?
boolean isBalanced(){
        return this.left.totalWeight() * this.leftSide ==
                 this.right.totalWeight() * this.rightSide &&
                 this.left.isBalanced() &&
                 this.right.isBalanced();
}
```

Fig. 2. Sample method in class `Complex` for a program that models a hanging mobile

Here we encountered a problem: Java and most object-oriented languages do not support equality comparison based on the value of data, and so the design and evaluation of tests in this context becomes a daunting task. We have solved this problem by designing the ***tester*** library [7], [8], [9] that compares any two objects by the value of their fields, traversing deeply to the primitive components, detecting circularity of data definitions, and making the test design simple and straightforward.

When the tests are evaluated, the student may choose to pretty-print all data fields defined in the `Examples` class, the class that represents the client to the student code, and to print either all test results, or only those that have failed.

```
IMobile big =
      new Complex(1, 6, 6,
                  new Simple(1, 15, "blue"),
                  new Complex(2, 6, 4,
                              new Complex(3, 2, 4,
                                          new Simple(1, 10, "green"),
                                          new Simple(3, 5, "red")),
                              new Simple(2, 20, "red")));
// Tests for isBalanced
boolean testIsBalanced(Tester t){
    return (t.checkExpect(simp1.isBalanced(), true) &&
            t.checkExpect(comp1.isBalanced(), true) &&
            t.checkExpect(big.isBalanced(), true) &&
            t.checkExpect(comp2.isBalanced(), false));
}
```

Fig. 3. The examples of data and tests for a program that models a hanging mobile

Libraries for Novice Programmers: World Game Library. One may ask, what kind of programs can students write in such a simple environment? Well we can design binary search trees, programs that represent cells in a spreadsheet that refer to other cells with formulas that need to be evaluated, build recursively defined lists of items, thus implementing a stack data type, tennis tournaments, etc. But to support design explorations and to motivate students, we have also built a ***world*** library [7] that allows students to program the behavior (the model and the display) of a graphics-based interactive game. Students extend the `World` class by adding fields that represents various game objects. They define the methods that represent the actions in response to the timer or a key press, producing a new instance of a changed world, and the methods that produce the scene that represents the current state of the world. The library creates the game canvas in a new frame, installs the necessary event listeners, and provides event handlers that invoke student-defined methods.

We can accomplish a lot with simple tools. The three libraries: ***FunJava*** that provides a novice-friendly simple language, the ***tester*** library, that makes the test design, method evaluation, and data display easy, and the ***world*** library that turns simple programs into interactive graphics-based games provide the infrastructure where student's focus is on the program design, free of idiosyncrasies and complexities of professional programming languages and libraries.

The great advantage of this approach is that the students learn to program in a truly object-oriented style from the beginning. They understand the dynamic dispatch of methods. We insist that every method handles only one task and delegates to helper method any complex tasks that arise (*the chain of responsibility principle*). Students have to reason about which class needs to be responsible for every task (i.e., where should the methods be defined), and they have to write examples of method invocation with the expected outcomes (tests) for every method they define.

Fig. 4. The snake game (by Matthias Felleisen). The snake moves on each tick, changes the direction in response to the arrow keys, looks for food, grows with each food eaten, avoiding the walls or itself.

With this foundation, we are ready to discuss more advanced ideas of program design and introduce programming language features that enable the design of reusable libraries. Each new programming language feature is introduced in the context of solving a problem encountered earlier: the way to eliminate code repetition, the way to handle problems that cannot be solved using purely functional style, the way to eliminate the need for excessive saving of intermediate results, etc. The framework for this stage of the curriculum is the study of designing *abstractions*.

4 How to Design Libraries

A novice has a hard time learning a number of features of modern object-oriented languages that have been designed to help a seasoned programmer to work effectively. It is important to present every language feature in the context of compelling examples that illustrate the reason for introducing that feature. Once our students mastered the basics of the program design in the object-oriented style, we focus on the design of abstractions that leverage different language features to avoid code repetition and to build reusable code. This provides a context for learning how libraries are designed and used. The ***design recipe for abstractions*** provides a systematic way to examine where abstraction is possible, to define what needs to be done, and to verify that the abstraction correctly accomplishes the desired task.

Abstract Classes. In the introductory weeks students have seen several classes (`Circle, Square, Rectangle`) that implement the common interface `Shape`. Each class included a field that represented the location of the shape in some *Canvas*, and it included methods that compute the area of the `Shape`, it distance to the origin, and a method `isSmallerThan` that compared the area of **`this`** `Shape` to the area of the given `Shape`. The code repetition in these classes is obvious, and it is easy to motivate the need for an `abstract` class that defines all common fields, includes a

constructor that initializes them, and contains a concrete implementation of the common methods as well as those that are common to most subclasses.

The other side of this coin is the introduction of our `abstract World` class that provides the entire functionality for designing an interactive graphics-based game, leaving to students the task of implementing the abstract `onDraw` method, and overriding the stubs of the `onKey` and `onTick` methods. This is their first encounter with a library. It provides an environment for designing interesting applications while focusing only on the design of the model.

Function Objects. *Java Collections Framework* includes several interfaces that specify functional behavior. The most commonly used ones are the `Comparable` interface and the `Comparator`. The only role the `Comparator` plays is to provide a wrapper for a method `compare` that compares two objects of the same type. In many functional languages, a function can be passed as an argument to another function (*functions are first class values*), but Java designers did not provide for this. Thus the programmer needs to define a class that implements this interface, design the needed method, define an instance of this class and pass that as the argument to the methods like `sort` or `findMin`, or `findMax`. However, rather than introducing these interfaces, we start with interfaces `ISelectBook` or `ISelectPerson` that implements a predicate that selects the objects with the desired properties and is used by methods like `findBook`, `containsPerson`, etc. The reason is to delay introducing the type parameters, and to show the students both the definition of the interface and the design of the classes that implement it. One example we use is to select all runners in the Boston Marathon that are female under 40 years old, all masters runners (over 50) etc. It is clear that we do not want to design the same `selectRunners` method several times, when the only thing that changes is the selection criterion.

We do mention that a similar technique is used for defining the action that the computer performs in response to the GUI button press or when an event handler is activated by the event it is listening to.

Mutation (State Change). We introduce the assignment statement and the resulting change of the state of a variable once the students are comfortable with designing classes, designing methods, and they understand the dynamic dispatch of the methods. We present problems that either cannot be solved without mutation (or some additional language construct), or where mutation simplifies the work to be done. One cannot design the data that represents students enrolled in courses, when the course data contains the list of currently enrolled students, and the student data contains the list of courses student is enrolled in, without changing the values of the data fields after they have been defined. A bank record representing an account needs to be modified when a deposit or a withdrawal is made, so that every program that has access to this data sees the change. The variable that holds the user's response to a question will only get its value once the user responds.

A survey of typical textbooks and papers describing introductory curricula shows that only a handful of them pay any attention to systematic design of tests from the beginning. We attribute this to two problems: the design of tests in the presence of state change is quite complex, and the design of test for mutation-free programs

requires support for extensional equality tests. Yet, designing programs that are not tested is a very bad habit to learn. Our experience has repeatedly shown us that even seasoned programmers make trivial mistakes in the simplest program components and that these are either very hard to find, or go undetected for extended periods of time.

Our introduction to state change comes hand-in-hand with the design of tests: with the setup of needed data, method invocation, testing of the effects of the method, and the reset of the data that has been used. The purpose statement for the method changes to include the word *EFFECT* where necessary. We defer until later the use of methods that combine the state change with returning a new value.

This is also the time when we begin to discuss the difference between two objects that represent the same value and two names for the same object. By now students are comfortable with the basic program design, the language syntax, and they can appreciate the subtleties of the data representation and aliasing.

Program Integrity and Usability. At this point students are ready to think of programs that will be used and modified by others. We introduce several techniques a programmer can use to expose the program behavior while hiding and protecting the internal details of implementation. We talk about the *visibility modifiers*, show how *constructors* can provide several different ways for the user to instantiate objects and to verify that the data satisfies the desired constraints (month is one of 12 possible values, hour does not go beyond 24, etc.), and introduce the *exception handling*.

Another important topic we begin to discuss is the definition of *equality* and the implementation of methods that compare two objects. Are two lists the same if they refer to the same instance? Or are they the same if their respective elements refer to the same instances or just represent the same values? The need for detecting circularity in data also comes into play.

Parametrized Types. Students see that we have been defining similar methods for data collections of different data types: binary search trees of persons, cities; lists of books or songs, etc. Even the function objects were targeted for only one type of objects. Having seen this, the introduction of generics (parametrized types) is a welcome new abstraction in spite of the complexities of the necessary syntax.

Abstracting over Traversals. All along we also present examples of methods that represent traversal over the items in the collection of data. We design methods `isEmpty()`, `Data getFirst()`, and `Collection getRest()` for both lists and binary search trees. Abstraction over the collection of these three methods introduces a new interface, `Traversal`, that represents a functional iterator:

```
Interface Traversal<T>{
  public boolean isEmpty();
  T getFirst();
  public Traversal<T> getRest();
}
```

We see that the methods that manipulate the collections of data can be defined outside of the class definitions of these collections. We have come a full circle: starting from standalone functions in the functional language, to designing methods that rely on the dynamic dispatch for selecting the appropriate action, to moving the methods to the

`Algorithms` class that deals with an arbitrary data, as long as the data collection provides the necessary hooks.

Loops, the *Java Collections Framework* `Iterator` interface, and the `Iterable` interface are introduced at this time.

Abstract Data Types. With this background we introduce the `ArrayList`, the `HashMap`, the `Stacks` and `Queue`, and other classes in the *Java Collections Framework*. We ask the students to implement the `Stack` and the `Queue` interface; design a mutable linked list, and use them in the context where one or the other can be used interchangeably. The *Depth-First Search* and the *Breadth-First Search* over graphs differ only in the way we implement the data set that keeps track of the next set of edges to consider.

When introducing the hash maps, we revisit the issue of equality. We show how to correctly override both the `equals` method and the `hashCode` method. Using the *JUnit* test framework, reading and writing the *Javadoc* style documentation are the last couple of steps for students to be ready to fully use the standard Java libraries.

Java Collections Framework. With students' knowledge of the meaning of interfaces for defining the behavior of data, abstract classes for implementing the common behavior of a union of similar data types, the use of function objects to define functions that algorithms can use, the introduction of the *Java Collection Framework* is very straightforward. Students understand the design, can reason about the implementation, and can implement some of the library classes themselves.

We complete the work with several discussions of the **resource management issues**. Memory usage, time-complexity of algorithms, the cost of using structural recursion, all are made visible through an assignment where students evaluate stress test runs. The classical data structures and algorithms are presented only to illustrate the design choices: indexed data structures make binary search possible, key-value associations allow for fast data lookup, linked lists allow localized modification of the structure, quicksort leverages the divide and conquer strategy, etc. Through simple programming assignments we show students the different ways how information can be represented as data: students manipulate images by modifying image pixels, they process text data computing word frequencies, they use our simple sound library to generate sound effects and background music for their games.

5 Summary

The *Program by Design* curriculum evolves. The *Bootstrap* component is building a web-based programming environment [10], the second edition of HtDP includes support for client-server computing over the network [3]. We have piloted a library that supports the design of applications for mobile devices. The tester library is a foundation for the development of a comprehensive software testing curriculum.

The curriculum has been used in many settings (after-school programs, summer camps for children, secondary schools, universities). Teachers of children who completed the *Bootstrap* program wonder at their improved math grades. Secondary school students who started with the *Program by Design* curriculum do well in the Advanced Placement in Computer Science (AP) test, even though the AP curriculum

follows a more traditional programming curriculum. Our university added a required course for the graduate Master's of Science program that is based on the *Program by Design* curriculum, to improve advanced student's program design skills.

Acknowledgments. The *Program by Design* is a work of the team led by Matthias Felleisen, with Matthew Flatt, Robby Findler, and Shriram Krishnamurthi its co-founders [7]. Kathy Gray has contributed to the design and initial implementation of the ReachJava segment [6, 11]. Kathi Fisler has worked on the further development of the curriculum. Emmanuel Schanzer is the designer of the *Bootstrap* component [2], [14]. Erich Neuwirth inspired the development of the sound library [9].

The two grants by the National Science Foundation (Redesigning Introductory Computing: The Design Discipline, DUE-0618543 and Integrating Test Design into Computing Curriculum from the Beginning DUE CCLI 0920182) provided partial support for the development and dissemination of this project.

References

1. Barnes, D.J., Kölling, M.: Objects First with Java: A Practical Introduction using BlueJ. Prentice Hall / Pearson Education (2008)
2. Bootstrap Project, http://www.bootstrapworld.org
3. Dann, W.P., Cooper, S., Pausch, R.: Learning to Program with Alice, 3rd edn. Prentice Hall, Englewood Cliffs (2012)
4. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs. MIT Press, Cambridge (2001)
5. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs, 2nd edn., http://www.ccs.neu.edu/home/matthias/HtDP2e/index.html
6. Felleisen, M., Findler, R.B., Flatt, M., Gray, K., Krishnamurthi, S., Proulx, V.K.: How to Design Classes, http://www.ccs.neu.edu/home/matthias/htdc.html
7. Findler, R.B., Flanagan, C., Flatt, M., Krishnamurthi, S., Felleisen, M.: DrScheme: A pedagogic programming environment for Scheme. In: Hartel, P.H., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 36–388. Springer, Heidelberg (1997)
8. Klaeren, H., Sperber, M.: Die Macht der Abstraktion, B. G. Teubner Verlag, Wiesbaden (2007)
9. Neuwirth, E.: http://sunsite.univie.ac.at/musicfun/MidiCSD/
10. Proulx, V.K.: ReachJava Libraries, http://www.ccs.neu.edu/javalib
11. Proulx, V.K.: Test-Driven Design for Introductory OO Programming. SIGCSE Bulletin 41(1), 138–142 (2009)
12. Proulx, V.K., Gray, K.E.: Design of Class Hierarchies: An Introduction to OO Program Design. SIGCSE Bulletin 38(1), 288–292 (2006)
13. Riley, D.D.: The Object of Data Abstraction and Structures Using Java. Addison Wesley, Reading (2003)
14. WeScheme, http://www.wescheme.org/