Experience with Teaching Program Design

Mirela Djordjević
Manhattanville College, Purchase, NY, USA

Mirela.Djordjevic@mville.edu

Abstract

Conventional approach to teach Java is to teach language syntax by expressing computations as sequences of assignment statements that change the value of variables and fields (instance variables). We follow a nontraditional approach to teach Java by teaching the program design, which place into the center classes and the design of classes. This approach requires the introduction of many classes of data and the development of several functions for each class and how Java's syntactic constructs support the program design. The purpose of this paper is to share our experience in teaching Program design following some pedagogical patterns: spiral mode, early bird, test tool, replication. The paper clarifies some colloquial or conventional characteristics of teaching Computer Science programming courses for Liberal Arts. Pedagogical patterns are shown with some details of use in our practice. Our experience shows the continuation of a long term progress of teaching "hard" programming courses but making them more attractive to our students.

Keywords: Teaching introductory programming course, Java, Program Design.

Introduction

Compared with many other subjects, Computer Science is not straightforward to teach. "Programming courses are generally regarded as difficult, and often have the highest dropout rates. It is widely accepted that it takes about 10 years of experience to turn a novice into an expert programmer" (Robins, Rountree, & Rountree, 2003). Considering that students view programming courses as very demanding and hard to comprehend, instructors are always challenged in the process of teaching computer science courses. Introductory programming courses are especially hard to teach, as those should set good foundations for higher level courses in computer science. The Program by Design curriculum has as its goal to provide a systematic introduction to the fundamentals of computing and programming.

Learning and Teaching

"Programming involves formulating mappings from the problem domain (via intermediate domains) into the programming domain – a process which requires knowledge of both the structure of the domains and of the mappings between them. Loops, conditionals, arrays and recursion have all been identified as language features that are especially problematic, and could benefit from particular attention" (Robins, Rountree, & Rountree, 2003). How do students learn these language constructs? Are loops, arrays and recursion placed in the center of the introductory course?

The body of CS knowledge is constantly changing, so we should consider a question: what should be taught in the introductory programming course as the foundation computer science education. Our traditional approach in teaching introductory programming course, say in Java, is to cover variables, assignments, if statements, methods, classes, loops and array list. Do we teach just Java language constructs and apply them to specific domain area (mathematics, calculations, drawing)? How do we focus on programming language syntax? Does the language syntax explain modeling of program design needs for any specific solution? Do we take these language constructs as extensions of the language, or they have profound role in building projects. In teaching object oriented programming languages we need to follow the model: represent data and relevant data connections, and functions.

Mville Experience

Manhattanville College (Mville) is a liberal arts college with computer science major that has an academic community of 1,200 students. Almost one tenth of the student body is enrolled in an introductory programming course before they graduate. While some students take introductory programming courses simply due to the school requirements, others take it for fun. Only a small number continue to major in computer science. This is a diverse group for whom the introductory programming course in Java is "the hardest course in their college career", which requires them "to think in a completely different way". In addition to the diverse student backgrounds, the highly technical nature of programming makes it difficult to master. To design, implement, compile and test even a small program requires a wide array of knowledge and skills – ranging from the syntax and semantics of the programming language, to the problem solving strategies and software design principles. Further, they have to gain a proficiency in the use of computer software tools such as editors, compilers, and debuggers.

Recently, three years ago, we started teaching Java by first introducing Program design with drRacket and Java. The **Program Design** approach that we used in our introductory programming course focuses on the idea of workshops ("Teach Scheme Reach Java", n.d.). Basic concepts in the introductory programming course are: variables, functions, conditionals, structures, lists and recursion. The "hard" topics: assignments, loops, array lists are postponed for the upper level course, while basic inheritance is introduced naturally by following design recipes. 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 level, apply equally well in a more complex context

"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." (Proulx, 2009.)

In this approach, we teach basic topics: functions, data structures, conditionals, mixed data structures, and lists with drRacket. Then we teach limited Java where a **class** can implement only one **interface**, and there are only two statements: **if** with a required **else** clause, and **return** expression. Our students get the understanding of designing classes that contain fields that are instances of another class, unions of classes, self-referential data, and mutually referential data. Functions from drRacket become methods in Java, and the object that invokes the method (this) becomes just an additional argument the method consumes. Because we do not teach mutation, the out-

come of every method depends only on its inputs. So it is always required to check that the outcome of a method invocation produces the desired value.

Program Design

Our understanding is that the main goal of introductory programming course should not be to teach any specific programming language, but basics of computing, or basics of program design, together with the stress on testing as a main way of approaching problems. These basics can be established through a disciplined approach and through "design recipes" (Proulx, 2009). A great deal of our curriculum focuses on understanding how information can be represented as data and how a piece of data can be interpreted to describe the information encoded within. Computing can be represented as a program that consumes data and produces new data according to a formula in the introductory level. Complex data are combined with programs which cannot be separated from data. Understanding data and how information can be represented as data, and how data can be used, deserves a main place early in the curriculum. Well-structured data defines numerous algorithms for extracting new information, and provides the context for learning the foundations of program design.

Design recipes structure program design as a series of steps. The steps should be followed whenever preparing a program. Those usually do not depend on a programming language, but it is recommended to teach functional programming without state changes that might be hard to test and follow. Rather than changing the state of variables, our programs produce new values at each step. This type of programming (functional programming) has the advantage that the outcome of each function depends only on the values of the arguments, not its position within the program.

Design Recipe for Designing Functions

Functions are seen in mathematics. Each function takes one or more arguments and produces a new value. It is easy to define the expected behavior of such functions. The *design recipe for functions* takes advantage of this functional behavior:

1. Analyze the problem by representing relevant information as (simple data or data structures).

2. Write down the purpose statement for the function, identifying clearly what it consumes and what it should produce. Write this information formally as a function domain and range, and accordingly write the function header.

3. Make examples of the data the function will consume, and the expected results. Prepare as many examples as needed to understand the problem.

4. Write all available data and functions already defined that could be used in defining new function. This includes fields of complex data and functions that consume data of the type of data available to us as the function arguments.

5. Now design the body of the function. If the task looks complex, divide it into smaller parts and make *helping* functions to solve the problem. (These functions will be designed later, following the same steps. For now we assume they have been defined already.)

6. Convert the examples from part 3 into formal tests and run them. Fix errors and test.

Design Recipe for Data Definitions

The *design recipe for defining data* guides students through the following process:

1. Check how problem information can be represented? Can it be one simple piece of data: a number, a symbol, or a String?

2. When several related pieces of information are needed to describe one object, combine them into drRacket structure (later these become Java classes). Define a drRacket data structure with one field per piece of information. For example, a student with a name represented as a String, and a year of birth as a number.

3. If the information refers to another complex object, use containment. This will be a structure or a class with a field that is an instance of another structure/class: A student with a name, year of birth, and a course that a student takes (with a title and grade).

4. If the information consists of several related objects with some common features but with several variants that distinguish them, define a union (students are freshmen, sophomore, junior and senior). Then define each member of the union separately. (In Java these turn into classes that implement a common interface or extend an abstract class.)

Design Recipe for Designing Abstractions

Once the programs become sufficiently complex students begin to see that certain sections of code appear again and again, perhaps with a slight variation. This motivates the design of abstractions. For examples, a list of students is just a list of objects. The design recipe for abstractions provides the following guideline:

1. Compare two pieces of code that are similar, highlight their differences.
2. Replace each pair of differences with a parameter and rewrite the general solution.
3. Implement the solution to each of the original programs and run the tests.

The Teach Scheme component follows the textbooks: How to Design Programs (Felleisen, Findler, Flatt, & Krishnamurthi, 2003.) and Picturing Programs (Bloch, 2010.) covering the following topics: structures, mixed data structures, and lists. The Reach Java component follows the textbook How to Design Classes (draft) (Felleisen, Flatt, Findler, Gray, Krishnamurthi, & Proulx, 2010.).

Pedagogical Patterns Followed by Program Design

Our lectures follow Program design and also enforce Pedagogical patterns (Bergin, J. 2000-2006.), such as the Spiral model, Early bird, Tool box, Mix Old and New, Grade it again.

Early Bird pedagogy suggests organizing the course so that the most important topics are taught first. Teach simple data and structures together with functions applied on those in drRacket, and then in Java is a way of setting the "big idea" first.

Considering the **Spiral Model** pedagogy, topics in a course are divided up into fragments: variables, functions, structures, lists. These fragments are introduced in the same order first in drRacket and then in Java that facilitates student problem solving. Many of the fragments introduce a topic, but do not cover it in detail. Just enough detail is given initially so as to form a basic under-

standing that can be applied to problem solving. Additional cycles in Java contain reinforcing fragments that go into more detail on the topic.

The intent of **Tool Box** pedagogy is to let the students follow design recipes in drRacket early in a course for use in later in Java part of the course. We well thought out our examples as they can be implemented in drRacket and Java. Students spent their early part the semester that they would need it to achieve mastery in Java.

Students examine and use computing artifacts much earlier than they can design and build them themselves, so the pattern **Larger than life** is followed as well. The course ends with topics: abstract classes, class hierarchy that allows showing Java libraries to be explored. We introduce drawing libraries to be used with shape classes.

Students practice design at all levels by following design recipes for each of the topics. **Student design spirit** pattern gives them quick feedback and peer review on early attempts. This pattern has to be enforced for each of the design recipes.

Considering **Test Tube**, students can answer their own "What if…" programming questions by providing the examples of the problems before doing any definition coding. They need to predict the results and place them explicitly in order to gain the good understanding of the problem and have a better approach to defining the functions in drRacket or classes in Java.

Students can often learn a complex topic by building several small parts of a larger artifact for **Fill in the Blanks**. This happens also whenever students make the wish lists of functions (methods) that complete the projects. This takes time to be

Pattern **Grade It Again** should be practiced to provide an environment in which students can safely make errors and learn from them, permit them to resubmit previous assignments for reassessment and an improved grade.

Our Experience

We have tried different approaches in our introductory programming courses: teach Java with stress on graphics/drawing examples, so domain knowledge is more easily understandable for our students, compared with traditional problems from mathematics; teach Alice with basic constructs (variables, assignments, if statements, loops) and then switch to Java. Both of these approaches are considered as traditional approaches and for most students' very hard course.

We introduced design program discipline to establish a valid process in problem solving by following design recipe steps and successful ending with the required tests. Our classes are enrolled usually by ten, twelve students and we cannot guide any valid statistical analysis on this data to prove the significance of this platform. Our experience shows that our students are more conforming to this approach but still think that this introductory course is not easy and they recommend it only to those students who want to continue computer science major. Their comments on the course are: "Good class, difficult at times", "A difficult course for many, but I enjoy the class and assignments very much!", "The course itself is interesting but requires a lot of logic", "I only recommend this class to students who have an interest in computer science because it's not an easy class."

Our experience shows the continuation of a long term progress of teaching "hard" programming courses but making them more attractive to our students. We will continue to teach basics of computing by program design with the stress of testing as a main way of approaching the problems. Our standing is that students' solid understanding of representing relevant information as formal data helps students design programs.

References

Bergin, J. (2000-2006). *Fourteen pedagogical patterns.* Retrieved from http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2000/2000_Bergin_FourteenPedagogicalPatterns.pdf

Bloch, S. (2010). *Picturing programs.* Retrieved from http://picturingprograms.com/

Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2003). *How to design programs*. Retrieved from http://htdp.org/

Felleisen, M., Flatt, M., Findler, R. B., Gray, K. E., Krishnamurthi, S. & Proulx, V. (2010). *How to design classes* (draft). Retrieved from http://www.ccs.neu.edu/home/matthias/htdc.html

Proulx, V. (2009). *Introductory computing: The design discipline*. Retrieved from http://www.ccs.neu.edu/home/vkp/Papers/ISSEP2011.pdf

Robins, A., Rountree, J., & Rountree, N. (2003) Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137–172.

Teach Scheme Reach Java. (n.d.) Retrieved from http://www.teach-scheme.org/Overview/

Appendix

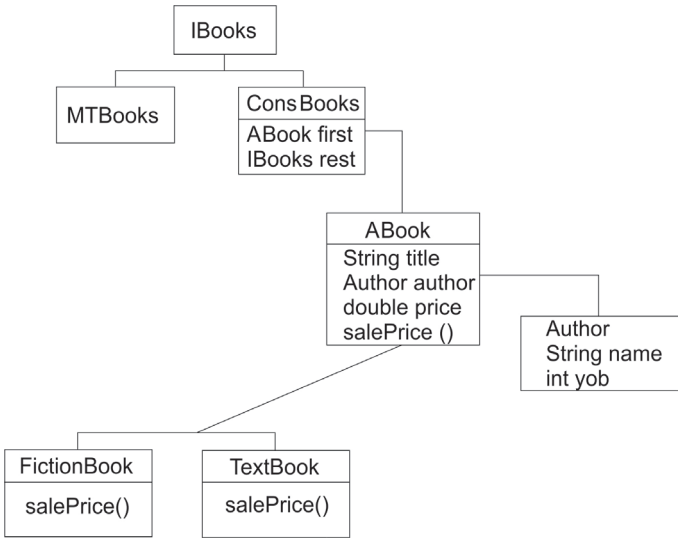
Some examples of data used in this introductory course: a student's record with the list of courses the student is enrolled in; book store's representation of textbooks, fiction books; ancestor trees; real estate's representation of offerings, the representation of files and directories; list of songs by some artists; list of images.

Project example with a book structure and an author structure, represented as drRacket program:

```
;; A book consists of title(string), author(string), price(number),
kind(string).

(define-struct book (title author price kind))

;; An author consists of name(String) and yob(number).

(define-struct author (name yob))

;; Examples of authors

(define eh (make-author "Hemingway" 1900 ))

(define ml (make-author "Liosa" 1935 ))

(define mf (make-author "Mathias" 1970))

;; Examples of books

(define oms (make-book "Old Man and the Sea" eh 10 "fiction"))

(define bg (make-book "The Bad Girl" ml 20 "fiction"))

(define htdp (make-book "How to Design Programs" mf "textbook"))

;; List of books

(define listbooks (cons oms (cons bg empty)))
```

Examples of functions: compute discounted sale price for the given book and given discount. Compute the total price of all books in the given list of books, total sale price of all books in the list, produce a list of books sorted by their sale price.

Java interfaces and abstract classes are naturally introduced with this project. The project in drRacket is easily transformed in Java classes as shown by the following diagram:



Corresponding Java code follows:

```
class Author {
    String name;
    int year;
    Author ( String name, int year){
        this.year = year;
        this.name = name; }
    /*   this.name    String
     *   this.author  Author
     *   this.year    int     */}
abstract class ABook implements IBook{
        String title;
        Author author;
        double price;
        String kind;
    public ABook(  String title, Author author, double price, String kind) {
        this.title = title;
        this.author = author;
        this.price = price;
        this.kind = kind;}
     /*  this.title String
     *   this.author Author
     *   this.price  double
     *   this.kind   String    */}
class FictionBook extends ABook{
 FictionBook (Author author, String title,  double price, String kind){
       super(author, title, price, kind);}
```

```
 public double salePrice(){
        return this.price * 0.5;}}
class TextBook extends ABook{
 TextBook (Author author, String title, double price, String kind){
             super(author, title, price, kind);}
 public double salePrice(){
        return this.price * 0.8;}}
```

DrRacket and related Java topics are listed in the Table 1.

Table 1: Basic Concepts

DrRacket	Java
Images, simple data, operations on simple data and images	Classes
Functions on images, functions on numbers, strings	Methods for simple classes
Structures, functions on structures Structures containing structures	Data definitions: classes, containment
Mixed data structures, conditionals	Unions, Interfaces
Lists, functions on lists	Class hierarchies, Interfaces

Biography

Mrs. **Mirela Djordjević**, PhD, Associate Professor, is a member of the Department of Mathematics and Computer Science and has been affiliated with Manhattanville College since 2002. Dr. Djordjević received her B.S. from The University of Belgrade in 1981, her M.S. from The University of Belgrade in 1986, her M.S. from The University of Maryland, Baltimore County in 1993 and her Ph.D. from The University of Maryland, Baltimore County in 1997.