A Concepts-First Introduction to Computer Science

Kenneth J. Goldman
Computer Science and Engineering
Washington University
St. Louis, MO 63130
kjg@cse.wustl.edu

Abstract

We present a unique "concepts-first" curriculum that exposes students without programming experience to the intellectual depth and breadth of computer science through hands-on experience with software development. The curriculum is supported by JPie, a tightly integrated programming environment that enables live construction of Java applications through direct manipulation of graphical representations of programming abstractions. The curriculum, projects, and classroom experience are described.

Categories & Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education, curriculum.*

General Terms:

Human Factors

Keywords

Concepts-first curriculum

1 INTRODUCTION

Virtually all entering college freshmen have used software, but most have not thought deeply about how it is constructed. Many students, including intended computer science majors and those with prior programming experience, have misconceptions about the field. They often assume that computer science education is not about ideas and creativity, but is instead primarily about learning technology "buzzwords" and the syntax of programming languages. As a result of these misconceptions, many students assume that they are not interested in the discipline, making it difficult to attract a diverse population to the field.

This paper describes a "concepts-first" computer science curriculum that exposes students to the intellectual depth and breadth of the discipline through hands-on experience with software development. Like a CS0 course, this course has a broad scope. However, in its delivery, it is more like a CS1 course, in that all major topics are reinforced by implementation projects.

In planning this curriculum, we recognized that simultaneously satisfying both objectives (broad coverage and software development experience) in a single semester would be unlikely if a traditional textual programming language were used as the vehicle of exploration. Moreover, many non-majors simply are not interested in learning syntax, and would much rather learn concepts and gain experience without up-front technical training. If students could directly manipulate programming abstractions, rather than encode them textually, we could side step the syntax difficulties and move directly into the ideas. Furthermore, if the programming environment allowed programs to be modified while they are running, students could learn more easily through experimentation. Finally, if the programming environment supported a standard model of computation, then students who continue in a standard computer science curriculum could transfer much of their knowledge and experience.

Our concepts-first curriculum is supported by JPie [8,9], a tightly integrated programming environment designed to make software development accessible and attractive to a wider audience. JPie enables live construction of Java applications through direct manipulation of graphical representations of programming abstractions. JPie supports delivery of our curriculum by making software construction more direct and efficient, thereby enabling students to concentrate on higher-level issues.

Using JPie, we have been able to offer an introductory course for non-majors (and potential majors) that provides students with both an overview of the field and the chance to experience the excitement of software development. The course exposes students to a wide range of ideas, and also lets them discover what it feels like to apply those ideas and make things happen in software. Moreover, we are able to accomplish this in one semester with a reasonable student workload.

The remainder of the paper is organized as follows. Section 2 provides background on the JPie programming environment. Section 3 describes the curriculum, which is subdivided into four main parts: fundamental abstractions, software design, algorithms and data structures, and concurrency and communication. In Section 4, we describe our classroom experience with the curriculum. Section 5 concludes the paper.

2 THE JPIE PROGRAMMING ENVIRONMENT

JPie is a tightly integrated programming environment for live software construction in Java. JPie treats programming as an application in its own right, providing a visual representation of class definitions and supporting direct manipulation of graphical representations of programming abstractions and constructs. Exploiting Java's reflection mechanism, JPie introduces the notion of a *dynamic class* that can be modified while the program is running, thereby eliminating the edit-compile-test cycle. JPie elevates the unit of discourse from the character to the semantic unit. Rather than encoding program abstractions as a linear sequence of characters, students can directly manipulate the abstractions. This, together with its support for live software modification, provides a more natural and fluid software development process that both raises the level of abstraction and eliminates many of the common pitfalls that beginning Java programmers face [10].

JPie differs from other integrated development environments, including those specifically targeted for computer science education [1], in that it supports live development and represents programs graphically rather than textually. However, JPie is not a visual language [3,5]. Instead, its visual representation serves as a front-end for Java. Consequently, JPie benefits from years of accumulated research and experience in programming language design. JPie programmers learn to work within a standard object-oriented programming model and leverage the entire Java API, enabling a smooth transition into textual programming and making live software construction in JPie a viable alternative to textual programming in introductory courses.

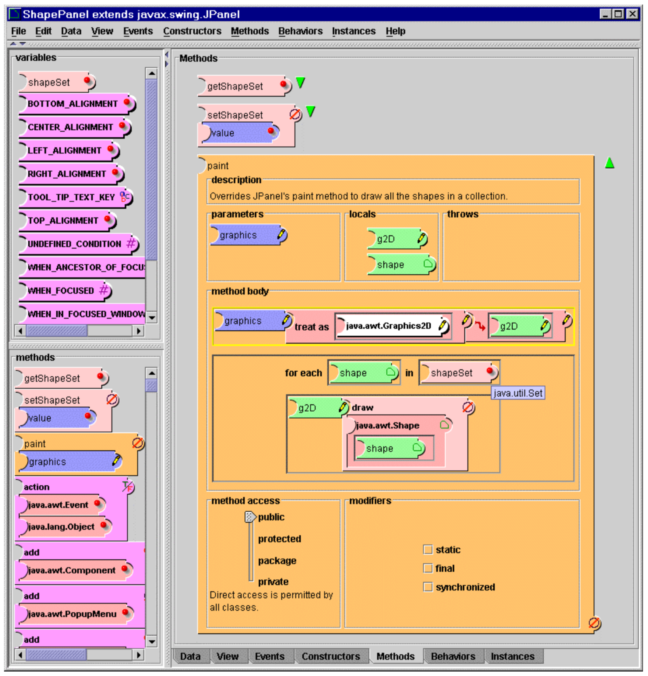
JPie's principal visual unit is the *capsule*. Capsules represent variable declarations, variable accesses, properties, methods, method calls, constructors, and constructor calls, and can also contain constants and expressions. Every capsule has a textual identifier, an icon to indicate type, and a color to indicate scope. JPie maintains consistency (of names, parameter lists, etc.) between each declaration and all of its uses.

Each class has a separate window, as shown in Figure 1. Tabs along the bottom reveal panels containing Data (instance variables), View (for graphical user interface construction), Event handlers (listeners to the view components), Constructors, Methods, Behaviors (periodic tasks that run as separate threads), and Instances (a list of instances of the class, selectable for viewing). In addition to these semantic regions, the window provides convenient summary lists of the variables and methods of the class (with inherited members shown in a different color).

Programmers manipulate capsules and other objects within clearly identified *semantic regions.* Programmer actions are understood by the system on the basis of the semantic region in which they occur. This allows declarations and other editing operations to be completed in one atomic gesture. For example, when a programmer drags a type capsule into the Data panel, the system declares an instance variable of that type (and automatically defines associated 'get' and 'set' methods). On the other hand, if the programmer drags that same type into the Methods panel, the system declares a method with that return type. Similarly, dragging an inherited method into the Methods panel creates a method to override the inherited one.

The class window in Figure 1 is for a dynamic class named 'ShapePanel' that extends Java's 'JPanel' class. This example illustrates the visual representation of variables, methods, formal parameters, method calls, actual parameters, casting, assignment, and modifiers. Overriding the paint method in this subclass of JPanel was accomplished by dragging the inherited method (from the summary list at left) into the methods panel. Upon declaration, the method *takes immediate effect*, even for objects that were already instantiated and whose paint method is called polymorphically by compiled classes. The user interface prevents the formation of syntactically incorrect statements and expressions, and provides immediate type-checking feedback.

JPie programmers can create threads in the conventional way, by creating a subclass of the 'Thread' class and defining a run method. In addition, JPie's Behaviors panel provides streamlined support for creating threads that carry out periodic tasks. For example, an animation might have a behavior to periodically


Overrides JPanel's paint method to draw all the shapes in a collection.

Direct access is permitted by all classes.

Figure 1. A dynamic class window in JPie

change the image. Each behavior looks like a method with a void return type, but has additional regions in which to specify a rate expression and a termination condition. Behaviors can be started automatically upon object instantiation.

JPie provides a thread-oriented debugger that uses the same visual representation that is used in the class windows. When a breakpoint or error condition is reached, the debugger shows the call stack as a series of tabbed panes. Each pane shows the visual representation of the method (or other item) responsible for that stack frame. The debugger highlights the expression that is currently executing (or about to execute, in the case of the top stack frame). The programmer can control the execution speed and watch the execution unfold, or can single-step through the execution expression by expression, with pop-up text displaying values for executed expressions. Editing in the debugger produces the same immediate effects as in the rest of the system.

3 A CONCEPTS-FIRST CURRICULUM

Our curriculum is organized around a series of 'big ideas." These are presented to students, along with supporting concepts, in a brief lecture at the beginning of class, and then students explore the ideas through laboratory exercises that they carry out collaboratively or independently within JPie. This section outlines the topics of the curriculum and briefly describes some of the seventeen projects and tutorials that reinforce the ideas.

Those accustomed to teaching introductory courses using traditional textual programming environments may find this curriculum remarkably ambitious. However, due to JPie's live development process and high level of expressive power, the curriculum fits comfortably in a one semester course that meets twice weekly for 90 minutes, and in which all programming projects are completed during class time.

3.1 Fundamental Abstractions

The first section of the course, approximately two weeks, introduces fundamental concepts of modeling, naming abstraction, types and values, classes and objects, methods and delegation, procedural abstraction with parameters and return values, sequential and conditional execution. The distinction between active and reactive computation is also introduced.

The students' first lab introduces classes as definitions of types of objects, and provides experience with modeling real world objects by representing their properties as values stored in instance variables. Students are handed a collection of childrens toys and are asked to list properties of each type of object from two different perspectives, that of a customer and that of a physicist. The students learn that the abstraction created depends upon one's point of view. For each physical object, students identify the property name, type, and value. They then use JPie to define classes whose objects encapsulate the properties they identified.

The next two labs introduce sequential and conditional execution, as well as active and reactive control. Students implement a clock that actively updates its time once per second, and they then implement a Body Mass Indicator in which the BMI formula is computed and updated interactively as the user adjusts the values of sliders indicating height and weight. Conditional execution is used to display the BMI value in a different color, depending upon the range in which the value falls. In addition to introducing computational statement, expressions, and control flow, these labs introduce the separation of the data model from the view seen by the user, as described in Section 3.2.1.2.

Reactive control is introduced in the BMI lab, where students must specify what should happen when the user manipulates the sliders. In JPie, each event handler is a listener method in the Java event model. The programmer demonstrates the user event of interest (mouse click, mouse entered, etc.) by performing the event on the selected component. Then, the user selects the desired event from a list of the recorded events. The listener, method, which is edited dynamically as any other method, is automatically registered to be invoked whenever the specified event occurs in a view of any instance of the class. This streamlined event handling mechanism exposes the execution model, but minimizes the programming mechanics so students can concentrate on the important ideas.

3.2 Software Design

Design is introduced as a creative process that affects the functionality, usability and efficiency of software. Design topics most emphasized include a separation of concerns, effective use of type systems, and design for correctness. Efficiency is discussed later, when algorithms and data structures are covered.

3.2.1 Separation of Concerns

Separation of concerns is emphasized as a way of managing complexity. Students solve problems by breaking them into smaller pieces, each of which can be understood independently.

3.2.1.1 Encapsulation

Encapsulation is a natural discussion topic in object-oriented programming. We emphasize that each object has responsibility for managing its own data, and that the methods provide an abstraction barrier around the data. Encapsulation is used from the start, first as a way to provide abstraction, and later as a way to design correct software.

3.2.1.2 Model/View Separation

Part of understanding modeling involves the realization that what is represented internally can be distinguished from the way it is presented to the user. This point is made through projects in which students construct graphical user interfaces and link them to the data model in their objects. In the clock lab, for example, students see the distinction between the variable that holds the current time, and the text property of the graphics component on which it is displayed.

In JPie, the View and Events panels of the class definition window support user interface construction. As in most GUI builders, one can drop graphics components into the Views panel to specify how each instance of the class will appear graphically. Component properties can then be connected to the properties of the class (i.e., instance variables) or to properties of other components in the view by chaining their capsules together. This very direct mechanism allows students to appreciate the idea of model/view separation with a minimum of overhead.

3.2.1.3 Local Coordinate Systems

Another effective way to emphasize the separation of concerns is through the use of multiple coordinate systems in graphics applications. We explain that each individual component "wants" to be concerned only with its own local coordinate system, and not necessarily with the coordinate system of its container. Furthermore, an application "wants" to model its data in real-world coordinates, which are then mapped by the application to the pixel coordinates on the screen. When the students implement a simple version of the game of pong, they model the application in a 1000x1000 coordinate system, but they must understand the mapping between the world coordinates and pixel coordinates of the window in which it is displayed. For example, when handling mouse motion events to move the paddle in the game, the students must convert from the window system coordinates to the world coordinates of the game. In addition, the Pong project provides experience with constructing multithreaded applications.

3.2.2 Type Systems

Beyond the basic notion that a class is a type of object, we provide opportunities for students to learn the importance of class hierarchies in understanding types and implementing software.

3.2.2.1 Class Hierarchy Design

The first exposure to class hierarchies is a design project in which students are presented with a specification for a software system in paragraph form. Their job is to identify the various types in the description, distinguish between the "has-a" and "is-a" relationships described, and then create a set of classes in a hierarchy that model the natural relationships. The specification we provide is very rich in relationships and gives the students an opportunity to think about alternative designs. In the course of this project, we encourage good design practices, such as pushing functionality as high as possible in the hierarchy, for inheritance by the classes below.

3.2.2.2 Inheritance and Specialization

Immediately following the class hierarchy design lab, we introduce the idea of overriding methods to achieve specialization. We provide students with an already built Calculator class that provides a graphical user interface of calculator buttons and display, but that does nothing when the buttons are pressed. The students extend this class to create a SmartCalculator class that overrides the various functions of the Calculator class to actually perform the desired computations. The calculator project also provides an opportunity to reinforce the concept of boolean logic, which is introduced in the textbook, by providing calculator buttons that perform various boolean functions.

3.2.2.3 Polymorphism

Polymorphism is one of the most powerful and beautiful features of object-oriented software design. However, students often need to see it several times before truly appreciating its value. To introduce polymorphism, we decided to take full advantage of JPie's support for live software construction. In particular, JPie allows methods to be overridden on the fly, with existing instances affected immediately. We devised an "animated characters" project, in which we provide a Sprite class that displays an animated gif image. The students then invent a hierarchy of subclasses of sprites, some that move around the screen in straight lines, some that oscillate, some that disappear when coming in contact with other sprites, etc. They create instances of these classes and, while the program is running, override or change methods to see how the sprites change their behavior. For example, they see that the "step" method is being called repeatedly on each sprite, but the resulting behavior is different depending upon the sprites specific subtype.

3.2.3 Program Correctness

Even students without programming experience have encountered "bugs" in progr ams, and they are fascinated to understand how these arise and how people try to prevent them. In the "bank account" project, in which we introduce the concept of a representation invariant that is established initially by the constructor and preserved by all methods. Students create their own assertion checking method that takes a boolean expression as a parameter and throws an exception when the value is false. They use this assertion checker to verify a representation invariant of their bank account class (namely, that the balance is never negative), as well as to check preconditions on the parameters (for example, that the amount being transferred is not negative). They then learn about testing by creating a test program that instantiates several bank accounts and attempts various operations (both legal and illegal).

When an exception occurs that is not explicitly caught or thrown by a method, the JPie debugger provides the programmer with the opportunity to modify the program and "try again" from that point, or catch (or throw) the exception and resume execution. This allows beginners to program for the common case, and add exception handling as issues arise. This way, students remain focused on the central task, rather than becoming overwhelmed by having to think about all the possible exceptions in advance.

Additional correctness issues surface in the algorithms and data structures section of the course.

3.3 Algorithms and Data Structures

Algorithms and data structures are central to computer science and an important part of any introductory survey course. Designing and analyzing complex algorithms is beyond the scope of this curriculum, but it is important for students to have a good understanding of what algorithms and data structures are, how they are used, and what kinds of building blocks used to design them. Algorithm and data structure efficiency are discussed, but the students are not asked to do any complexity analysis.

3.3.1 Iteration and Recursion

To introduce iteration, we ask the students to create a color gradient, a two-dimensional raster of colors where the red color component increases across the x-axis and the green component increases down the y-axis. Next, the students use nested loops to implement a "15 -puzzle." We provide the students w ith a bitmap image, which they load into an image raster. Their application treats the raster as a 4 by 4 grid of tiles, a designated "empty" tile that they paint black. They then construct a method to swap tiles in the raster, and add an event handler that swaps the empty tile with the tile under the cursor when the user clicks the mouse. Finally they add constraints so that the swapping occurs only when the empty tile is in a neighboring square. Their implementation is done entirely in terms of pixel operations on the raster. The issue of multiple coordinate systems also resurfaces here, as students must relate coordinates within a tile to the entire raster, as well as the coordinates of tiles to the mouse location in the graphics component.

Recursion is introduced with a project called "Persian Recursion" that is based on a paper from the mathematics literature [4]. Students, again using the raster, recursively subdivide the image by painting lines in various colors, using a color function that produces images reminiscent of Persian rugs.

The JPie debugger provides proactive support for detecting common logic errors before they become fatal errors. This includes dynamically adjustable stack bounding to detect infinite recursion, and dynamically adjustable loop bounding to detect infinite loops.

3.3.2 Use of Fundamental Data Structures

Because JPie does not yet provide data structure visualization support, we currently do not ask students to design data structures in this course. However, they do use Java's built -in TreeMap to implement a dictionary project in which the user can interactively define and look up words. Other types of simple data structures (such as linked lists and arrays) are covered in textbook exercises.

3.3.3 Persistence

As a second part of the dictionary project, students learn about persistent data and how programs save data in files. They modify their dictionary program to save the dictionary and then reload it in the constructor when the program starts. To avoid file format issues, the students write to the file using an ObjectOutputStream, letting Java do the serialization automatically.

In a later project, students explore databases as a way to store persistent data on behalf of programs. They implement a "mu ltiplayer adventure game" in which the entire class interacts with a single shared database that maintains the game state. Using special support in JPie for treating relational databases as collections of objects, each student (or pair of students) implements a game client, all sharing one persistent data store.

3.4 Concurrency and Communication

A modern survey of computer science would not be complete without some treatment of concurrency and communication.

3.4.1 Synchronization and Deadlock

The importance of synchronization is discussed in the context of multithreading and representation variants, which are introduced early in the course. Then, students implement a version of the classic dining philosophers problem [7]. They implement a Philosopher class, a Chopstick class, and a Table whose constructor instantiates chopsticks and philosophers and arranges them in a virtual ring around the "table." The students run the program until it hits a deadlock situation. They experiment with aborting a thread to see that the rest of the philosophers resume eating. They are then challenged to modify the Table constructor so that deadlock cannot occur in the system. They are asked to explain why their solution prevents deadlock, but we do not expect a formal proof.

This project is enabled by JPie's deadlock detection support. When JPie detects a deadlock, a separate window appears with a visualization of the cycle in the wait-for graph. Within that visualization, the programmer can click on individual threads involved in the cycle to see them in the debugger, and optionally terminate their execution to break the deadlock.

3.4.2 Interprocess Communication

With so much prior exposure to the internet, students are naturally curious about how electronic communication occurs. We spend some class time explaining IP and TCP, and then the students implement an "internet chat" application using sockets to pass messages between the client and server that they implement. Their server is multithreaded, to accept multiple clients, and they implement a ClientHandler class that is instantiated by their server each time a new client connects. Students then, using IP addresses directly, connect to other students servers and exchange messages. Their user interface displays a history of the "chat" with each user's messages identified by name. JPie's on -the-fly exception handling eases the construction process because students can program for the common case, without having to anticipate all the possible I/O exceptions. When exceptions do arise, they can be handled and execution continued.

3.5 Textual Programming

The last project of the semester introduces students to textual programming. Using a feature in JPie that generates textual source code that corresponds to the graphical representation, students define a simple class and see a corresponding textual implementation. We ask students to modify their program in JPie and see the changes in the textual source code. Finally we ask them to save the textual source code to a file, compile it, run it, and then make some simple textual modifications to see compiler errors, as well as to change the behavior of the program. Since students already understand the abstractions (classes, objects, methods, parameters, etc.), their first encounter with textual source code appears to be more enlightening than intimidating.

4 CLASSROOM EXPERIENCE

This curriculum has been implemented as "Washington University CS123: The Computer Science Way of Thinking," [11] and is currently in its second semester. It is taught as a seminar-style laboratory course, in which students do all programming projects during supervised sessions. Homework consists of reading assignments and short written exercises from a computer science survey text [6] that provide a general context in which to understand the programming projects. Students are encouraged to collaborate in pairs on the projects, and by listening to their conversations we are able to monitor their thought process as they work through the projects. We admit only students without prior experience so that there is no intimidation factor among peers.

The course has been successful in attracting a more balanced gender mix (approximately 50% women), supporting findings that women are likely to find a top-down (i.e., concepts-first) presentation of the field more appealing than a bottom-up approach in which technical details dominate. [2]

After learning the initial mechanics of drag and drop within JPie, students move quickly into thinking about the projects at a conceptual level. Through the sequence of projects, various features of the programming environment must be introduced, and their mechanics explained, but student questions indicate that they are thinking deeply about the higher level ideas, and not just the mechanics of completing the projects. Students generally complete projects within the allotted time, and there is enough flexibility in the course that students can finish projects in the following class period if necessary. After the first offering of the course, we have rearranged some topics. In particular, we now introduce the concept of class hierarchies early in the semester, since they figure so prominently in object-oriented design.

5 CONCLUSION

We have presented a concepts-first curriculum that provides a broad introduction to computer science, with hands-on software development experience in a live interactive programming environment. We are interested in working closely with educators at other institutions, both undergraduate and K-12, who would like to use this curriculum in their classrooms.

6 ACKNOWLEDGEMENTS

I thank the following students for their contributions to the JPie project: Joel Brandt, Ben Brinckerhoff, Vanessa Clark, Melanie Cowan, Matt Hampton, Adam Mitz, Jonathan Nye, Sajeeva Pallemulle, and Richard Souvenir. I also thank the students in CS123 for valuable feedback, and Ben Brinckerhoff, Adam Mitz, and Jason Froderman, who have been teaching assistants for the course. This work was supported in part by the National Science Foundation under CISE Educational Innovation grant 0305954.

7 REFERENCES

1. David J. Barnes and Michael Kölling, *Objects First with Java: A Practical Introduction Using BlueJ.* Prentice Hall/Pearson Education (2003).
2. Lenore Blum, "Transforming the Culture of Computing at Carnegie Mellon," Computing Research News, November 2001.
3. A. Borning, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory." ACM Transactions on Programming Languages and Systems (1981), vol. 3 pp.355-387.
4. Anne M. Burns, "Persian Recursion." *Mathematics Magazine*, vol. 7, 1997. pp. 196-199.
5. T.D. Kimura, J.W. Choi, and J.M. Mack, "A visual language for keyboardless programming." Technical Report WUCS-86-6, Washington University in St. Louis, June 1986.
6. Nell Dale and John Lewis, *Computer Science Illuminated.* Jones and Bartlett Publishers, 2002.
7. E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes." *Acta Informatica* (1971), vol. 1, pp. 115-138.
8. Kenneth J. Goldman, "A Demonstration of JPie: An Environment for Live Software Construction in Java," to appear in the OOPSLA'03 Conference Companion, October 26-30, 2003, Anaheim, California, USA.
9. Kenneth J. Goldman et al., "JPie: Programming is Easy," http://jpie.cse.wustl.edu, July 2003.
10. Kenneth J. Goldman, "An Interactive Environment for Beginning Java Programmers." Washington University Department of Computer Science and Engineering, August 25, 2003, *submitted for publication.*
11. Kenneth J. Goldman, "Washington University CS123: The Computer Science Way of Thinking," http://www.cse.wustl.edu/~kjg/cs123, January 2003 .
12. Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books: New York, 1980.