Teaching Introductory Programming to IS Students: Java Problems and Pitfalls

Mark O. Pendergast
Florida Gulf Coast University, Fort Myers, FL, USA

mpenderg@fgcu.edu

Executive Summary

This paper examines the impact the use of the Java programming language has had on the way our students learn to program and the success they achieve. The importance of a properly constructed first course in programming cannot be overstated. A course well experienced will leave students with good programming habits, the ability to learn on their own, and a favorable impression of programming as a profession. In this paper I detail how and why the Java programming language was selected for our curriculum, how my teaching has evolved over the years since Java has been adopted, and what successes and failures I've encountered. Specifically, this paper discusses some of the peculiarities of the Java programming language that make it difficult to learn and some suggestions to overcome them, as well as some of the teaching paradigms and programming tools I have employed. What my experience has shown is that combining the use of a modern interactive development environment such as the Netbeans, with active learning and a breadth-first approach is found to increase student satisfaction, increase success rates, and lower dropout frequencies.

Keywords: Java, Information Systems, Education, Introductory Programming Course, Syntax Errors, IDE.

Introduction

Like many things in life, first impression matters, and programming is no exception. The importance of a properly constructed first course in programming cannot be overstated. Such a course leaves students with good programming habits, the ability to learn on their own, and a favorable impression of programming as a profession. A poor experience may result in a "just get by" attitude, bad programming habits, and could lead to a change in majors. Ala-Mutka Uimonen, and Järvinen (2004) assessed programming style as a function of modularity, typography, clarity, independence, effectiveness, and reliability, whereas Reddy (2000) defines style constructs specific to the Java language. Instilling in students these habits and providing them with an enjoyable first experience in programming is important.

Since 2001, enrollment in the course *Introduction to Business Programming* has shrunk by 50% at my institution. This combined with the 50% attrition rate of students who enter *Introduction to Business Programming* and of those who complete the course *Intermediate Business Pro-*

Editor: Kam Vat

gramming has made the task of determining the factors that lead to success (or failure) in learning to program all the more important. Because of this our department is reviewing its curriculum, the technology used in courses, and recruiting strategies. Part of this assessment is determining the proper amount of emphasis to place on the programming skills taught to and expected of our graduates and the proper way to teach these skills.

During the boom of Computer Science/Information Systems (CS/IS) in the 1990's the loss of students to other disciplines was not a major concern. These students were just written off as not having an "aptitude" for programming. Nevertheless, this philosophy will change owing to shrinking enrollments in CS/IS fields (Zweben & Aspray, 2004). Hensel (1998) foretold of shrinking enrollment, predicting that enrollment would drop as a result of a decline in college age students, poor preparation for technical classes (leading to failures and reluctance to enroll), and changing interests in students. This added to fears of outsourcing, the dotcom bust, and the slow down in hiring after Y2K conversions have kept students away from CS/IS. Indeed, CS/IS faculty members have little control of national birthrates and the hiring practices of corporations, but we can look for ways to attract more students and retain the ones we now have.

This paper examines the impact the use of the Java programming language has had on the way our students learn to program and the success they achieve. The first section provides some background on why Java was selected as the programming language in our program. This is followed by an enumeration of the features and characteristics of Java that have presented particular problems to our students and a review of paradigms and tools used by other educators to overcome such problems. The structure for the *Introduction to Business Programming* course is then described along with assessment of outcomes. The paper concludes with a discussion of lessons learned and the results achieved.

The Move to Java

Teaching programming courses to Information Systems (IS) students in the College of Business has much in common with teaching Computer Science (CS) students but there are some differences in student background and prerequisite courses. Unlike CS students, IS students are generally only required to have a semester of statistics and a semester of "business calculus" to graduate. More often than not, these courses are put off to their senior years. Therefore the mathematical background of IS students consists of what they may or may not have learned in high school and what they may or may not remember of their required learning. In addition, a typical IS major only takes 20-30 credits of computer related courses whereas the CS major is required by ABET (Accreditation Board for Engineering and Technology) standards to have 42 or more hours of required CS study. This allows IS students to take only 2 or 3 courses whose primary purpose is to teach programming fundamentals. Although CS and IS students may have different educational backgrounds, their learning and problem solving styles are fundamentally similar. In fact, Sim and Wright (2002) found no difference in problem solving styles between CS and IS students.

As late as the 1990, COBOL was found to be the programming language of choice for IS students due to its popularity in business applications. Yet, in the early 1990's many IS programs began using the PASCAL language, besides COBOL to provide their students with the experience of procedural programming. This trend was reinforced by the introduction of personal computers and Borland's Turbo Pascal on campuses. Nevertheless, the coupling of Microsoft's introduction of Visual Basic and the advent of the world-wide-web (WWW) did cause many IS programs to abandon Pascal in favor of Visual Basic or Java towards the late 1990's. Furthermore, the increased need for developing students' programming skills in web-based programming (or scripting) languages such as Perl, JavaSript, and Java over traditional application programming languages like COBOL and C are expressed by Culwin (1999), as well as Noll and Wilkins (2002). See IS 2002 Standards (Association for Information Systems, 2002) for more information on IS curriculum standards.

On our campus we have proponents of all three languages: Visual Basic (VB), Java, and C++. We chose to teach Java because of its dominance in web applications, it is less complex and easier to debug than C++, and it has a syntax based on C/C++, making it easier for students to transition to those languages once they graduate. Tyma (1998) cited these reasons along with platform independence, higher programmer productivity, and better systems stability as reasons for moving to Java. Recently, NASA has chosen Java as its programming language for the Mars rover missions (Sun Microsystems, 2004) owing to its boosting programmer productivity. Java is a good choice for graphics courses because it is extensible and has a built-in graphics library (Bergin & Naps, 1998). The most recent TIOBE index of software programming languages places Java at the top of the list followed by C++ and C (TIOBE, 2006). Java has long been denigrated for its slow performance because of its use of a virtual machine to execute programs; however, the perceived performance difference may not be as great as some people claim. Mangione (1998) found little or no difference for most tests, and Lewis and Neuman (2003) found Java to be as fast or even faster than C++.

Java's Strengths and Weaknesses

Teaching object-oriented programming (OOP) and programming languages in support of object orientation (OO) has come a long way since Decker and Hirshfield (1994) enumerated their ten reasons why CS/IS programs were avoiding the teaching of OOP in their introductory programming course. Chief among these reasons were a resistance to change and the idea OO programming was "too hard." At that time OO programming languages were evolving and coming into and going out of favor rapidly. Languages such as C++, SmallTalk, Lisp, EIFFEL, and SATHER were all vying for the hearts and minds of programmers. According to (Kolling, Kock, & Rosenberg, 1995), none of these languages could be considered as appropriate for use by beginning programmers. Kolling went on to enumerate ten requirements for an object-oriented language that is to be used in a first year programming course (Please refer to Table 1).

Table 1: Java's Compliance with Kolling's 10 Criteria

Kolling Requirement	Java support
1. The language should support clean, simple, and well-defined concepts. This applies especially to the type system, which will have a major influence on the structure of the language. The basic concepts of object-oriented programming, such as information hiding, inheritance, type parameterization and dynamic dispatch, should be supported in a consistent and easily understandable manner.	No, since Java inherited much of its OO nature from C++, it inherited many of its pitfalls. Java did replace multiple inheritance with single inheritance augmented with a new construct called "interfaces". The use of single inheritance over multiple inheritance not only makes grasping an object hierarchy easier, but it removes the need to comprehend complex rules used to resolve ambiguous references under multiple inheritance.
2. The language should exhibit "pure" object-orientation in the sense that object-oriented constructs are not an additional option amongst other possible structures, but are the basic abstraction used in programming.	Yes, unlike C++, all code within a Java program must reside in an object. GUI and file IO are controlled by library object. However, this doesn't preclude someone from creating a poorly structured program.

3. It should avoid concepts that are likely to result in erroneous programs. In particular it should have a safe, statically checked (as far as possible) type system, no explicit pointers and no undetectable un-initialized variables.	Yes, unlike scripting languages, Java has a tightly typed system and the compiler automatically detects un-initialized variables. Though Java does have reference variables, it has no explicit pointers or use of indirection. The fact that students do not have to dispose of dynamically allocated variables removes many opportunities for errors. Java's strict type checking often seems restrictive to students, especially those with scripting experience.
4. The language should not include constructs that concern machine internals and have no semantic value. This includes, most importantly, dynamic storage allocation. As a result the system must provide automatic garbage collection.	Yes, Java is platform independent and does not contain constructs that concern machine internals (e.g. register variables). Variables have the same sizes and ranges across platforms. Java does provide automatic garbage collection. Not having to worry about memory leaks and removing the possibility of referencing objects that have been disposed of makes programming makes debugging much easier for students.
5. It should have a well-defined, easily understandable execution model.	No, in the case of over-loaded and overridden methods it is not always clear to beginning students which method will be invoked. Also the use of static methods and variables complicates understanding where data resides and how it can be accessed.
6. The language should have an easily readable, consistent syntax. Consistency and understandability are enhanced by ensuring that the same syntax is used for semantically similar constructs and that different syntax is used for other constructs.	No, again, due to its parentage with C and C++, there are multiple constructs for performing simple tasks, e.g. there are at least three ways to increment a variable by one: i = i+1; i++; and i+= 1; Confusion can be reduced by using consistent class examples that avoid the use of C++ short cut syntax.
7. The language itself should be small, clear and avoid redundancy in language constructs.	No, not only does Java inherit a large number of constructs from C++, it adds a large library of system objects that must be learned as well. Students respond well to the introduction of Java library objects when they are presented on an "as needed" basis and practiced in programming assignments. For example, spending an entire lecture listing all the Java GUI objects is not as productive as asking the students to learn a few each week as part of their homework.
8. It should, as far as possible, ease the transition to other widely used languages, such as C.	Yes, Java shares many language constructs with both C and C++. Java constructs such as the virtual machine, wrapper classes, and garbage collection have been adopted by Microsoft in their C#.NET and J#.NET languages. Making the transition to these languages relatively easy as well. The transition in the reverse direction is important as well. Many of my students have had C++ experience in high school and readily adapt to Java.

9. It should provide support for correctness assurance, such as assertions, debug instructions, and pre and post conditions.	Yes, to some degree. Java does provide an exception handling mechanism to assure proper execution and provides automatic testing for array bounds, null pointers, and arithmetic exceptions. Students find the use of Java exception handling constructs easier and more logical to use than pre-emptive "if statement" testing. However, consistency is a problem, in the case of arithmetic exceptions a divide by 0 in floating point operations results in a value of infinity, while in integer arithmetic the result is a divide by zero exception.
10. Finally, the language should have an easy-to-use development environment, including a debugger, so that the students can concentrate on learning programming concepts rather than the environment itself.	Yes, many mature IDEs are available, including Borland's JBuilder, Eclipse, Sun Microsystems Sun One Studio, and the NetBeans, an open source IDE. The main drawback to these environments is their cryptic compiler error messages and clumsy form editors. The latest edition of Netbeans has an improved forms editor and annotates syntax errors by underlining them in red. Students find it easy to learn the forms editor and the check as you type syntax error reporting is the same as they experience using popular word processors. Both Eclipse and Netbeans are free to the students.

Kolling et al.'s observations from 1995 are as relevant today as they were when they first expressed them. Accordingly, the next section presents what I consider to be Java's most difficult aspects for students to detect, correct, and comprehend. These language idiosyncrasies are included because they either do not occur every time, are hard to spot when desk checking code, are due to an inconsistency in the Java language design itself, or a combination of the above.

Java Language Frustrations

The very structure of a language can make it difficult to learn and teach (Pendergast, 2005). In that regard, the design of Java is no different from other compilation-based programming languages in that Java enforces strict type checking. Students who have some experience with scripting languages that have un-typed variables and fewer syntax rules will be especially vexed. Sun Microsystems designed Java to be platform independent and support numerous device types ranging from web servers to mobile devices like cell phones. This has led to an explosion of features and a standard library with more objects that can ever be addressed in a single programming course. In order to make Java more attractive to existing programmers Java was designed around C/C++ syntax, thus inheriting many of C/C++'s weaknesses. The following discussion details some of the most common/difficult aspects encountered by students in their beginning programming class. These areas of concerns were selected as examples because they either violate a general rule of the programming language, or are a result of an inconsistency in the design of the Java language and its object library. A coping strategy is thereby suggested for each concern to facilitate my students' learning the language

Extraneous Semicolons

Beginning students tend to take anything a professor or a textbook says literally. When their textbook (Liang, 2007, pp.21) states "Every statement in Java ends with a semicolon (;)", then that's what they do. The following code snippet includes five such cases:

```
public class SemiClass ; // 1
{
  public SemiClass(int parameter); //2
 {
    int i = 0, j = 0, x = 0;
    if(i < 9); //3
       j = 1;
    else; //4
       j = 2;
   for(x = 0; x < 100; x++); //5
   {
     i = i +x;
     return;
    }
   }
}
```

This code generates five syntax errors when compiled by the J2SDK, but only one of the erroneous semicolons is flagged. Extra semicolons at the end of *if*, *while*, and *for* statements will not generate compilation errors. Instead the code will compile, but not execute as intended since the extra ";" ends the statement. As a result, a student's *for* loop might execute a do-nothing statement 100 times, and their intended body of the loop only once! This is a very common error made by students early in the Introduction to Programming course. In particular, the addition of ";" after the ")" on *if* statements. The frequency of this error increased with the version of Netbeans that does syntax checking as students type in the program text. The students often type a correct *if* construct, then as they think about what should come next the compiler underlines the *if* statement in red and informs them of a missing ";". They then add the semi-colon, and type in the statement to be executed if the condition were true.

To help overcome this problem, it is suggested that textbooks should more precisely define what a statement is; instructors should use lots of examples during class presentation, and practice using a compiler or integrated development environment (IDE) that supports warning messages for control structures with do-nothing bodies.

Variable Scope Errors

In Java, as in C++, variable may be declared in the class definition, as parameters in methods, and as local variables in methods. Java allows local variables to be given the same name as those defined in the class. Furthermore, a method in Java may declare multiple variables with the same name as long as they occur in different code blocks. Java code resolves ambiguous references by using the variable declared in the block of code.

Another confusing aspect of variable scope is with variables defined in *do-while* blocks and in the header of *for* loops. In the example below, even though the *while* part of the *do-while* is considered part of the statement, the condition in the *while* cannot access variables defined in the *while* block. However, variables defined in the header of a *for*-loop can be accessed by code in the block.

```
      do{
         int x = 2;
         cost = cost + x;
         x = x-1;
       }while(x > 0);  // compile error here

       for(int i = 0; i < 3; i++)
       {
        int x = x+i;
       }
```

Variable scope problems can be overcome to some extent by using proper Java variable naming conventions. Under these conventions, local variables are defined with all lower case letters (e.g. productname) and class variables are declared with mixed case using lower case for the first word in the name and upper case for the first character of subsequent words (e.g. productName). This is just a partial solution because it differentiates only variable names that are more than one word long, and because naming conventions are not enforced by the compiler (and would be somewhat difficult to do so). A better solution would be to not allow local variable and method parameters to be given the same name as class variables. A very common example of scope errors is when students define a variable within a “try” block, then attempt to use it after the try block. Fortunately, the compiler does catch this sort of error.

For now, I have adopted the strategy of telling the students to never name a local variable the same as a class variable, and to declare all their variables at the beginning of the methods. This helps them spot naming conflicts, avoids ambiguous definitions, and makes the code easier to read

Integer Arithmetic Errors

These occur most frequently when an equation that yields a double result has an intermediate calculation that contains one integer being divided by another. Not only is this type of error hard to spot while desk checking it is also difficult to explain the rationale behind the standard. Students can readily understand that results must be truncated when they are stored into integer variables, but have a harder time with the concept of truncating numbers that are to be stored into double or floats. Example:

```
int x = 1, y = 2;
double z = x/y + 44.0;
System.out.println("z = "+z);
```

In this case z is given a value of 44.0 instead of 44.5. The best advice to give to students is to double-check every division for divide by zero errors AND integer truncation. Changing the operation of a compiler to use double precision math for all intermediate operations would simplify the teaching of the language, but may generate unintended errors in existing programs. A compiler generated warning message is more appropriate. This error occurs more frequently when students progress into the Intermediate Programming course where they must create programs that perform complex accounting and finance operations. Providing students with (or requiring them to generate) a good test data set helps in the learning process.

Handling NaN and Infinity Values

Next to grading, explaining language inconsistencies is my least favorite teaching task. This is particularly true of Java's handling of arithmetic operations errors. Java will generate an ArithmeticException when a divide by zero occurs for integer division, but not for double division. Instead, Java assigns the value of infinity to the double result. Taking a square root of a negative number does not produce an exception; instead, it assigns a value of NaN (not a number) to the result. The following example illustrates both cases.

```
   double y = Math.sqrt(-1);
   double x = 5.0/0.0;
   System.out.println("x = "+x+"  y = "+y);
   int z = 1/0;
   System.out.println("z = "+z);
```

produces:

```
  x = Infinity  y = NaN
  Exception in thread "main" java.lang.ArithmeticException: / by
zero at Main.main(Main.java:12)
```

To overcome these errors students must be instructed to carefully examine every equation, and to rely not on catching *ArithmeticException's*. Also, code must be added to test for results that produce infinity or NaN. The handling of divide by zero errors by the Java engine should be consistent, either always producing an exception or always result in infinity. As with the integer arithmetic errors, NaN and infinity problems usually crop up in students taking the Intermediate programming courses.

Operator Precedence Problems

One exam question I recently gave to my class required them to calculate the lines of code for a system using some of the classical empirical estimation models such as the Cocomo, Bailey-Basili, and the Boehm formulations (Pressman, 2005, pp. 660). Completing the exam question only required the ability to plug values into an equation and generate the result. I was surprised to discover that over half of the class could not do this correctly for an equation as simple as:

```
E = 5.5 + .73K1.16
```

Not only did they not know that the value of K should be raised to the 1.16 power with base 10 before being multiplied by .73, many did not know that the addition should be done last. They assumed everything should be done in a strict left to right sequence. While this particular class is not typical of all students at my institution or all Information Systems students, the problems they experienced pose a further teaching challenge. Correctly converting mathematical equations to Java code not only requires the understanding of Java precedence rules, but precedence rules used in the equations themselves. Business programming requires very accurate calculations of accrued interest, present and future values, and depreciation, as well as other items of interest. Accurate answers cannot be stressed enough.

Most, if not all authors of Java programming books include tables showing operator precedence order. Getting the students to learn how to correctly interpret the tables is a first step. A second step is to require extensive testing with problem sets that have known answers.

Zero Relative and One Relative Inconsistencies

Java, like C and C++, uses zero relative indexing for arrays. Most Java library classes also use zero relative indexes for retrieving items (e.g. Strings, Vectors, JComboBox, JList, JTable). Exceptions to this rule are found in the case of accessing fields in Java SQL objects. In particular the *PreparedStatement* and *ResultSet* objects are one-relative. Student programs which use one-relative indexing for arrays and container objects generally result in *IndexOutOfBoundExceptions* being thrown. *SQLExceptions* are thrown for using zero-relative indexing with *PreparedStatements* and *ResultSet* object produce *SQL Exceptions*. These exceptions are thrown only if the programs try to access every element; if not, then the error can go unnoticed.

Dealing with this inconsistency requires diligence by the students while coding and debugging. Having a consistent standard for the creation of standard language objects would help. Introductory students who have prior experience in languages other than C or C++ seem to have trouble adjusting to zero relative indexing of arrays and objects.

Casting and Type Checking

This is as much a conceptual problem as it is a programming problem. Many of my students have learned web-scripting languages, such as PHP, that do not require the declaration of variables and that do no type checking. They see this as a faster, easier, and therefore better approach to programming. Those of us with experience managing and maintaining systems have learned "the hard way" the value of languages that have tight type checking. Tight type checking reduces coding errors in both the creation and maintenance stages of a program's lifecycle. Students with only scripting experience do not have these preconceived notions. In either case, student must learn to understand type checking and casting in order to create Java programs.

Java's type checking makes perfect sense, once you understand it. Basically, Java will allow you to store values in variables without complaint as long as there is no loss of precision. Integers can be stored in longs, floats in doubles, integers into doubles, etc. However, loss of precision compiler errors will result if doubles are stored into floats or integers. When this is explained my students nod their heads indicating understanding, but still write code like the following:

```
double d = 3.0;

int x = d+1;
```

They assume that since d does not have a decimal part, it is plausible to be used with an integer. Since most of the functions in the Java Math object return doubles, the need to store doubles in integers comes up a lot. One solution that often appears in books is to cast the result as an integer.

```
int x = (int)Math.sqrt(143);
```

This has the effect of converting (and truncating) the result into an integer. So long as students realize they are truncating the decimal part of the answer and not rounding it, then their program will produce the intended results. However, often times they forget, or they assume that it won't make a difference in their final result. Therefore I also teach them to use the Math.round method. Also the students need to learn that casting can be used to convert real numbers to integers, but casting will not convert Strings or characters to numbers. For example:

```
char c = ‘5’;

int x = (int)c;
```

x will be set to the ASCII code for the character 5 (53), not 5.

Casting Objects

Beyond numeric type conversions, casting is a necessity when working with object streams, Java data structure, and container objects such as Vectors, JTables, JComboBoxes, and JLists. These objects maintain lists of other objects. An object of any type can be inserted into them without compiler complaint since the “add” methods accept objects of type “Object” (Object is the root type of all Java objects). Java allows objects of a subtype to be stored in a supertype without casting. The reverse is not true. Supertype objects cannot be stored into subtype variables without casting. Since the “get” methods associated with container objects return a value of type Object, casting is mandatory. The conceptual problem that students have with casting objects is they assume they are converting data as they did when casting primitive data types. Therefore they make mistakes like the following:

```
JTextField nameInput = new JTextField();

…

String s = (String)nameInput;
```

When they define the object *nameInput* they are creating a Java interface object that allows the input of string data. However, simply casting the object to a string does not retrieve its data. The example above will generate a compilation error since String is not a subtype of JTextField. This error can be explained and fixed. Casting of objects as they are retrieved from Vectors or other container/collection objects can produce run time errors that are less than obvious. In the following code a String object and an Employee object are stored into a Vector, they are both retrieved into Strings. The code compiles correctly but will generate a runtime error (ClassCastException) on the second call to “get”.

```
Vector v = new Vector();
String a = "A String";
Employee e = new Employee();

v.add(a);
v.add(e);

String s1=(String)v.get(0);
String s2=(String)v.get(1);//runtime error!
```

Avoiding errors of this sort is a matter of keeping track of the order and type of objects added to the Vector, and/or, by properly using the *instanceof* operator when retrieving items. The best advice to give to beginning students is to tell them to only add objects of one type to collections and containers. Since I require my students in the Introduction to Business Programming course make use of both the JCombobox and Vector objects, these errors do occur in their programs. Students in the Intermediate Programming course experience these errors when retrieving data from different columns in the JTable object.

The problems presented in this section illustrate that seemingly simple concepts can become very difficult to program if students do not have the proper level of comprehension of the syntax, structure, and sometimes-arbitrary/inconsistent rules of a programming language. An updated version of the Java language, know to some as Java 1.5 and to others as simply Java 5, makes certain constructs easier to handle, for example "boxing" primitive objects, enumerations, and generic typing (Austin, 2004). While this does make some code more readable it does little to alleviate the overall complexity and consistency of the language.

White and Sivitanides (2002) stipulate that both procedural and object-oriented programming require cognitive skills at the "formal operations" level. Students must possess the capability to deal with abstractions, solve problems systematically, and engage in mental manipulations. Once their level of cognition is exceeded then "burnout" insures (or as my students would say, their brains are fried). Students experiencing burnout are less likely to continue in a CS/IS program. Therefore it is imperative to create strategies to present information in a manner that will allow average students to succeed without experiencing a level of frustration that could possibly result in students' "burnout" mentality.

Paradigms and Tools

Dawson and Newman (2002, pp.126) put forward the notion of "empowerment" or the "act of enabling" as the ultimate goal for IS/CS education. They believe that "the most useful attribute we can give students is the confidence to find their own solutions to any given IT problem". Once students have attained the ability to learn something new from existing information, empowerment can be interpreted as the confidence to try new techniques, skills necessary to solve problems, and the ability to work effectively in a team. They go on to show that empowerment can be achieved by giving students a realistic and challenging task that stretches them beyond their experience and encourages them to reflect on what they have learned. But how does one take a student with no programming experience and get them to a level to where they can take on such assignments? What is the optimal mix of lecture and active learning? Should numerous small assignments be given to build their skills and confidence? Or should fewer more challenging assignments be given to allow them to experiment and learn with less time pressure? What tools and techniques should be used to handle the different needs of "left-brained" and "right-brained" thinkers? Should programming be taught depth-first or breadth-first?

Bruce, and others (2004) shed some light on this dilemma by identifying five learning strategies to be employed by students and techniques that can be used to progress students from the most basic strategy to more complex ones.

(1) Following – where learning to program is experienced as 'getting through' the unit.

(2) Coding – where learning to program is experienced as learning to code.

(3) Understanding and integrating – where learning to program is experienced as learning to write a program through understanding and integrating concepts.

(4) Problem solving – where learning to program is experienced as learning to do what it takes to solve a problem

(5) Participating or enculturation – where learning to program is experienced as discovering what it means to become a programmer.

By the time students graduate and enter the job market they should be using the participating or enculturation strategy for programming. It has been my experience that students cannot be expected to reach that level in the first or even second programming course. Therefore it is necessary to employ both curriculum and instructional techniques that enable them to progress in their learning styles. The Information Systems curriculum at my university starts students with an introduction to programming, followed by courses such as Intermediate Programming, Database, Data Communications, Systems Analysis, and ends with a Capstone Projects course. As the students advance through the course sequence they are expected to develop advanced learning methods, (refer to Table 2). During Introduction to Business Programming the students start with the ‘following” technique, often using in-class learning exercises or examples from the book to guide them while coding their assignments. As students gain more confidence and their skills increase they rely less on adapting examples and more on their own programming skills. Many exhibit this transition by adding features to programs that were not specifically required by the assignment (or even demonstrated in class), and by refusing help during active learning sessions. Intermediate Business Programming students start the course with at least a “coding” level of technique. The Intermediate course requires them to write web-based programs that access databases. This, in turn, compels the students to enter the “understanding and integrating” stage as they are required to integrate knowledge from the Database and Data Communications courses. The following sections detail teaching paradigms and tools used to help students progress from stage 1 to stage three of the Bruce model during their first two programming courses.

Table 2: Curriculum mapped to learning

Course	Techniques employed by students
Introduction to Business Programming	1) Following, 2) Coding
Intermediate Business Programming	2) Coding, 3) Understanding and Integrating
Database	3) Understanding and integrating
Data Communications	3) Understanding and integrating
Systems Analysis and Design	4) Problem solving
Capstone Projects	5) Participation

Metaphors, Demonstrations, and Active Learning

In order to empower my students in their study of Java programming , my students must learn three things almost simultaneously; they must learn Java syntax and structure, a sizable portion of the Java class library, and how to think in an object-oriented manner and convert those thoughts into code. Much of this is memorization of rules and exceptions to rules, too much of which leads to frustration and burnout. Astrachan (1998) demonstrates how metaphors and demonstrations (referred to by the author as hooks and props) can be used to help make complex concepts understandable and memorable. These include Dr Seuss’s *Cat in the Hat* book to demonstrate recursion and a “Frisbee” (Astrachan, 1998, pp. 22-23) in a plastic bag to demonstrate parameters passed by reference.

McConnell (1996) adds two other active learning techniques that could be useful for both introductory and intermediate programming. *Algorithm tracing* fits where students work together to predict how an algorithm will function based on different inputs, and *physical activities* (role playing) work where each student takes the part of a different object. The algorithm tracing technique could be combined with a lecture on using an interactive debugger. In this way students not only learn how to step through code manually (desk checking), but also learn to set break points, examine variables, and control program execution in the debugger. An easy to use interactive development environment (IDE) in the classroom is necessary to support this and other active learning activities.

In order to improve retention rates and make my teaching of java more effective, I changed from a depth first strategy to a breadth first strategy in my introduction to programming course. In the past students built applications that concentrated on the fundamentals of programming (variable definitions, control structures, OO design principles). These assignments had little or no GUI (graphical user-interface) elements, requiring only console mode of programming, involving text-based input and output (I/O). Under the "breadth first" approach I still cover basic java statements, control structures, class definition, basic GUI design, and file I/O. The assignments are designed to allow students to learn to build java applications using objects from the standard libraries (e.g., swing, io, util) using an interactive IDE. In depth theory on object-oriented programming is held off for a later course. I believe that creating applications with a standard GUI interface in the introductory course gives the students a sense of achievement and puts to the forefront issues of interface design and usability.

In addition to changing to a breadth first approach, I try to set student expectations in the courses by likening learning to program to learning to communicate in a foreign language. Students are cautioned not to expect to be fluent until they have completed the entire IS curriculum (advanced programming, database, networking, and capstone projects). I help set the expectations of students in the Introduction to Business programming course by explaining that they will be learning at the "words and phrases level". By that I mean they will gain the ability to read and understand Java code (or at least know where to look things up), and the ability to write small programs to perform explicit tasks (calculate mortgage payments, maintain lists of employees). Classes are taught in a computer classroom (28 student PCs and an instructor station) and consist of 50% demonstration (sample programs) and 50% active learning. Students are shown examples of programming concepts in the first half of the laboratory session, and then given a short program to work on in the second half. Example codes created in class are posted on the class web site each week.

Programming assignments are started in class where help is available. This enables students to learn from one another's mistakes and to progress past the inevitable stumbling blocks the new concept presents. In the beginning of the semester students rely heavily on my examples, many of them typing in the examples as I build them at the front of the room, thus displaying the "following" learning methodology. As the semester progresses students rely less on the in-class examples and more on their reasoning ability to create programs. This is evidenced by the questions they ask, fewer students ask, "what do I type next" and many ask, "why am I getting this error". Some students are confident enough to work ahead during the lecture portion of the class so that they can complete their assignments early. By the end of the semester students are comfortable with the syntax of the language and the development tools. They rarely ask how to fix syntax errors or which language construct should be used. This indicates that they have reached coding level (2) of learning and are ready to progress to the understanding and integrating level (3), according to the Bruce model (2004)

In my Intermediate Business Programming course they learn at the "short story and novelette" level. Here they have fewer, but longer assignments that can be considered a complete application (albeit one with limited features). For example, an application that maintains a product inventory database table and has the ability to perform searches and generate reports. During this course students are presented with technologies necessary in order to create applications in different environments. These include database access techniques (SQL) and introductory web programming (Java Server Pages). The assignments they create require them to integrate knowledge presented to them in this course, with knowledge from the introductory programming course, and the knowledge they are receiving in their database course (taken concurrently with intermediate programming). As in the introduction to programming, class sessions are split between lecture and active learning. Diagrams (in the form of rudimentary UML collaboration diagrams) are used to show how the various components of their programs interact with one another and the rest of the system. In this manner, level 3 methods of learning are incorporated into the course, requiring students to think at this level. Those students who have trouble making the transition still have their textbook and in-class examples to draw on; however, they quickly learn that these only present fragments of the picture and that they must work to integrate them.

Interactive Development Environments

If active learning is to be experienced in programming courses, then a suitable set of programming tools, including a debugger, needs to be available (Kolling et al., 1995). In selecting an IDE for course work there is a need to balance the amount of complexity of the IDE with the benefits of a full-featured environment. Cost is another factor. An IDE that is too expensive is unlikely to be adopted by the University and would not be affordable for students to install on their own PCs. Many textbooks come with free versions, evaluation versions, and stripped down versions of IDEs. Naugler and Surendran (2004) stipulate that the overhead of learning complex IDEs must be offset by their use in more than one course. They also recommend that if a complex IDE is used at the introductory level, then only a subset of its features must be used. Naugler and Surendran go on to caution about hidden costs of a product, academic neutrality of competing companies, and to emphasize formal education over tool-based training.

I've taught Java programming to students using a variety of programming environments. These environments have ranged from simple text editor/command line programs such as Textpad (© Helios Software) to fully integrated development environments such as Forte(© Sun Microsystems), JBuilder (© Borland), and most recently, NetBeans (© NetBeans.org). I've settled, for the time being, on Netbeans for several reasons:

- It is free to the students and university from http://www.netbeans.org. It requires little or no attention from our IT staff. As an open-source program it does not promote one company over another (Borland versus Sun).
- Tutorials on its use are available.
- It has features that ease learning, namely the visual identification of syntax errors (the lines of code with syntax errors are underlined in red, if you hover your cursor over the line the error is displayed in a window). Keywords and comments are color coded in blue and gray respectively.
- Visual cues are built in to help students find matching { }, [ ], and ( ). However, these cues don't seem to stop students from writing programs with extra or missing braces { }. This generally results in an obtuse compiler error message on a subsequent line, followed by every remaining statement in the program being flagged as erroneous.
- It has an integrated debugger (useful for intermediate and advanced programming courses).
- It has an integrated form editor, useful for visual programming. The value of this cannot be understated. The use of a visual editor for creating interfaces allows beginning students to create professional looking interfaces and reduces the amount of memorization of library classes.
- It pops up a hint window showing a list of possible selections and a definition of each. This helps to reduce the amount of memorization required.
- It supports the development of applications, applets, and server pages. This includes the Apache Tomcat server, useful for debugging servlets and Java server pages. Students are

not required to install and configure the server since it is run automatically from within the IDE.

- It as an optional mobility pack that can be added to allow the creation of Java programs for PDAs and cellular phones.
- It can be used throughout our curriculum as it supports database access, provides mechanisms for version control, JAR packaging, and Javadoc.

NetBeans is, however, a full-featured IDE that has many features not immediately useful (and even confusing) to beginning students. Yet, I've found that the students have little problem using NetBeans to create programs. In the beginning most adopt the "following" method of learning the Netbean workbench interface, that is, they write down and follow procedures I give them. As students become more advanced they find new features of the IDE and take advantage of them. -

There is a continuing, and often childish, debate within the Java community over which IDE is better, Eclipse or Netbeans. Proof of this is the 580,000 hits produced when Googling the phrase "Netbeans versus Eclipse". Falkman (2005) reviews the differences between the two IDEs, pointing out Eclipse's use of Standard Widget Toolkit (SWT) in place of Swing GUI objects. SWT objects rely on the GUI widgets native to the host O.S., allowing them to look and act like applications developed for that O.S. According to Falkman this was done for performance reasons but at the expense of using nonstandard classes. In addition, Netbean's use of Swing objects preserves look and feel across platforms and many companies have invested heavily in the Swing interface (Udell, 2002). Gallardo (2004) correctly points out Netbeans and Eclipse are feature for feature well matched. They are both extensible; therefore, any feature that one has can be filled in with third party plugins. Both have the ability to create projects, write and debug both web-based and traditional applications, perform version control, and package applications. Both have powerful corporate support, Netbeans is sponsored by Sun Microsystems and Eclipse by IBM. Currently, Eclipse is the market share leader for Java IDEs (Binstock, 2006). When I moved away from Borland's JBuilder in 2002 I wanted to adopt an IDE that would have an ease of use comparable to Microsoft's Visual Development Environment. A big part of this is an integrated forms editor. At the time, Netbeans (then known as Forte) had such feature, whereas Eclipse did not. Also, creation and testing of web-based applications is made easy in Netbeans since it has the Apache Tomcat web server bundled in. With Eclipse, using Tomcat means an additional download and install, plus the use of valuable class time to explain how to configure and manage a web server. Obviously, it is worthwhile to revisit the selection of an IDE each year and a change to Eclipse may be in our future.

Introduction to Business Programming Course Structure

As previously stated, I moved from a depth first to a breadth first approach to teaching the introductory programming class. In general, depth first courses stress teaching of core language constructs such as control structures, basic data types (up to arrays), and arithmetic statements, as well as basic algorithms. User interfaces are often "text only" with GUI elements being put off to later courses. Breadth first approaches may take many forms, but usually include basic GUI interaction; some data processing (arithmetic statements); and a limited set of file I/O. The implications for this are that students are able to create complete applications with GUI interfaces that perform file I/O, but they will be less proficient in using some of the basic control structures and data structures. This gap is filled in during the intermediate and advanced programming courses. The motivations for using a breadth first approach is to give students a broad view of what programming is during their first course (which is often when the students make their decisions to remain in a major program of study). Additionally, as the students are more likely to have a feeling of accomplishment when they complete a program which has much of the same look and feel of typical commercial applications they have used in the past.

In my course I use Daniel Liang's *Introduction to Java Programming, Comprehensive Version*, (Liang, 2007) and the Netbeans 5.0 IDE environment. The comprehensive edition was chosen over the core version of the book so that the same book could be used in both the Introduction and Intermediate programming classes.

Every assignment presents a application-driven approach employing a specific GUI development In the beginning the students create simple forms using the NetBeans graphical editor and a "null" layout manager to practice programming, doing application logic, and then back-end data resources. More complex interfaces using multiple panes, menus, and advanced layout managers (grid, border, flow) are covered in the second portion of the course. The final part of the course covers exception handling and file I/O. Students are evaluated using eight to ten programming assignments, three exams, and in class participation.

Course Outline

- Introduction to programming and basic computer architecture, role and responsibility of the programmer, ethical responsibility of system builders.
- Chapter 1 and Tutorial, Introduction to Java and Netbeans
    - Creating a project and an application
    - Creating a simple data entry form program (JFrame, JTextField, JLabel, JButton objects).
    - Compiling and running the program
- Chapter 2 Data types and operations
    - Arithmetic operators, operator precedence.
    - Assignment – simple calculator that adds, subtracts, multiplies, divides, raises numbers to a power (Math.pow), and performs modulo operations.
- Chapter 3,4 Control Statements
    - Stress *if, while*, and *for*.
    - Mention switch, do-while, and variations.
    - Assignment – use if statements and JCheckBox objects to determine if you are "Totally Cool".
    - Assignment – use for loop to calculate future value of an investment, $F = P(1 + r)^n$ for a range of years. Incorporate JTextArea for results.
- Chapter 6 Arrays
    - Stress declaring, creating, accessing array elements, and sequential search.
    - Use of Arrays object to sort and fill.
    - Mention advanced searching and sorting techniques
    - Assignment – Basic array operations, load an array with random numbers, provide operations for displaying the array (in a JTextArea), finding the average value, finding the smallest value, and sorting (using Arrays object).
- Catch-up and review, exam 1
- Chapters 12,15 Graphical User Interfaces (GUI) Programming
    - Stress JFrames, GridLayout, BorderLayout, FlowLayout. Controls JTextField, JLabel, JButton, JCheckBox, JRadioButton, JComboBox, JPanel, JTextArea, JScrollPane, JToolBar, JMenu elements.
    - Become proficient in designing containment hierarchies and using the Netbeans GUI editor

  - Mention other layout managers, advanced elements (JTable, JList, JTree), and how to create the interfaces without the GUI editor (using just a text editor).
  - Assignment – replicate the MS Windows calculator application scientific mode GUI.
- Chapter 5, Methods
  - Stress defining methods, calling methods, call by value, variable scope, and when to use methods.
  - Mention access specifiers, storage specifiers, overloading methods,
  - Assignment – create methods that calculate future value, mortgage payments, and future value of annuities. Incorporate these into a GUI interface to create a financial calculator.
- Chapter 7, Objects and Classes
  - Stress defining classes, proper naming conventions, defining variables and methods, toString(). Design of data storage objects e.g. Employee, Student, Course, Library-Book, etc.
  - Mention storage specifiers, access specifiers, static variables, and OO design languages such as UML.
  - Assignment – create an Employee class capable of holding an employee's first name, last name, job title, and wage. Incorporate the class into a form-based application capable of creating employee objects, and inserting them into a JComboBox, and deleting them from a JComboBox.
- Catch-up and Review, exam 2
- Chapter 17 Exception Handling
  - Stress Try/Catch/Finally clauses, Exception object hierarchy, checked versus unchecked exceptions. Demonstrate use of try/catch for handing NumberFormatExceptions generated by Integer.parseInt and Double.parseDouble.
  - Mention generating and throwing exceptions.
  - Assignment – Revise previous assignment to handle NumberFormatExceptions generated when handling employee wage data.
- Chapter 18 Input and Output
  - Stress Character versus Binary Streams, use of PrintWriter, BufferedReader, JFileChooser, File, ObjectInputStreams, and ObjectOutputStreams. Discuss basic sequence of operations for New, Open, Save, and Save-As operations.
  - Mention other forms of binary data files, parsing text input, advanced serializable features, and input file filters.
  - Assignment – modify Employee assignment to allow employees objects to be stored and retrieved from an ObjectStream (Open, Save, Save-As operations). Use JFileChooser, ObjectInputStream, and ObjectOutputStream objects.
- Review
- Finals week

A Typical Class

Class either meet once a week for a 2 hour and 45 minute session or twice a week in two 75-minute sessions. Classes are held in a 28-seat computer classroom. Each student has his or her own workstation (PC-Windows XP). As students arrive they login to their workstations and open both the class notes (in a web browser) and Netbeans. Most have this done before the period begins, the rest are able to get their stations ready during the opening class announcements. The period begins with a 15-20 minute introduction of a Java programming construct (the didactic portion). This is followed by the creation of a demonstration program to illustrate the concept and common errors that occur when using it. During this time most students will follow along on their workstations, interrupting with questions or to resolve problems. The students are then given a problem similar to the example just worked to resolve together in class. This active learning starts with the framing of the question, then one or more students will volunteer the procedure to solve the problem (pseudo-code), finally each student will attempt to create a program or procedure that implements the solution. This learning is then reinforced with a take-home assignment due the following week. If time permits the students are allowed to start their assignment in class and receive one-on-one help from the instructor. For example, a 75 minute period might be broken down as follows:

- Lecture: Show array declaration and creation concepts, review for-loop syntax. (20 minutes)
- Demonstration: Show a how to calculate the sum of the values of items in an array of doubles. (10 minutes)
- Active Learning: Have the students create a procedure to find the min (or max) value in the array, they are free to help one another or work on their own. (30 minutes)
- Assignment: Students create a program that loads an array with random numbers; provide operations for displaying the array; finding the average value, finding the smallest value, and sorting (using Arrays object). (15 minutes)

For each of my courses I create a website containing syllabus, assignments, class notes, links to online tutorials, and the sample programs created in class. Written comments on the course evaluation given at the end of the semester indicate that the information posted on the web is well received. Having the notes available online allows the students to actively participate in the lecture and enables them to follow along with the in class examples on their workstations. I chose not to use a Web classroom tool such as Angel (http://www.angellearning.com) or WebCT, instead the information is posted on a standard web page. Students use campus email or their own email to submit assignments. Students are encouraged to ask questions in class, in person during office hours, and via email. Since most of my students live off campus and I'm willing to answer email on weekends, email seems to be the preferred means to ask for help.

Assessing Outcomes

Assessing the Student

For grading purposes, students are evaluated on eight to ten programming assignments, three exams, and class participation. The course score is weighted 60% exams, 35% assignments, and 5% participation. The programming assignments are graded on a ten-point scale with about half of the points being awarded for correct function, the other half for programming style and user interface design. The exams are a combination of multiple choice and short essays. The multiple choice questions are designed to measure the understanding of Java's syntax and rules, the essays portion to see if students can complete short segments of code, identify errors in code, or explain how to solve a problem. Grades are a necessary evil, useful both for motivating the student and for recording their achievements so they can be later compared with their peers for jobs and graduate school admissions. The balance I find it necessary to strike is how much to weigh the assignment versus the exams. If not enough weight is given to assignments then they tend not to be turned in. On the other hand, I find it prudent not to give too much weight to assignments simply because it is not always possible to know how much of the work is done by the student and how much help was received from other students, or from examples found in books and on the internet. Exams allow me to test the depth of understanding of the material in the assignments and compel the students to study material not specifically required by the assignments.

In addition to the formal evaluation tools I've also found it useful for students to perform informal self-assessment of their work. I conduct this in the form of a simple survey following a project or team assignments. Surveys I have used in the past ask such things as: how much time was spent on the assignment; what was the most difficult task; what was the easiest task; what you would do differently next time. As one might expect, students list "starting sooner" as the primary thing that they would do differently next time. User interface design/building is usually listed as the enjoyable, and data management/handling the most difficult. Tables 3 and 4 present data from a recent group assignment given to the Intermediate Programming Class. In this assignment students worked in groups of 3 or 4 to create a program that automated the completion of the U.S. 1040EZ tax return. The program was required to gather the data from the user in a series of forms located on a tabbed pane. This involved determining if the taxpayer is eligible to use the form; collect personal information and income information; calculate the taxes and refunds using Tax Rate Schedules; store the results in binary format; and print the tax return. On the survey the students were asked to indicate which of the programming areas they perceived as difficult. File I/O and Data Management as the most difficult issues (reference Table 3). The data management was difficult for them since this was the first group project they had worked on that required each team member to create a separate GUI form (based on Java Swing library's JPanel) and share data objects between them. The key lies in the communication between the forms whenever a data item changes. Table 4 shows how the students allocated their time between the various lifecycle tasks.

Table 3
Student self assessment of difficulty

Programming function	% Indicating difficulty
File I/O	64%
Printing	21%
Tax Calculations	21%
Data management	36%
User Interfaces	7%

Table 4
Time allocation for the project

Task	Average % time spent in task
Project management	11%
Version Control	9%
Programming	43%
User Interface Design	11%
Debugging	15%
Final testing	9%
Other	2%

Perhaps more revealing (and more useful) than grades would be to track the student progress through (Bruce, et al., 2004) five levels of learning (Refer to Table 2). Since the progression is done across a series of courses it would be necessary to employ a self-assessment instrument at the end of each course. Informal indications of progress can be seen through the questions students ask and the assignments they turn in. For example, at what point are they able to correct their own syntax errors, do they experiment and add features not required by the assignment, do they ask for direction in implementing advanced feature. Invariably students who earn an A or B in the Introduction to Business Programming course have successfully progressed into level 2. Those earning a C most often are students who have not progressed beyond stage one, but have put in the effort to struggle through assignments. Figure 1 shows the percentages of students getting an A or B each term, Note the disparity between fall and spring semesters. During the fall semester the course is taught in the evening. This has two consequences, first, a large percentage

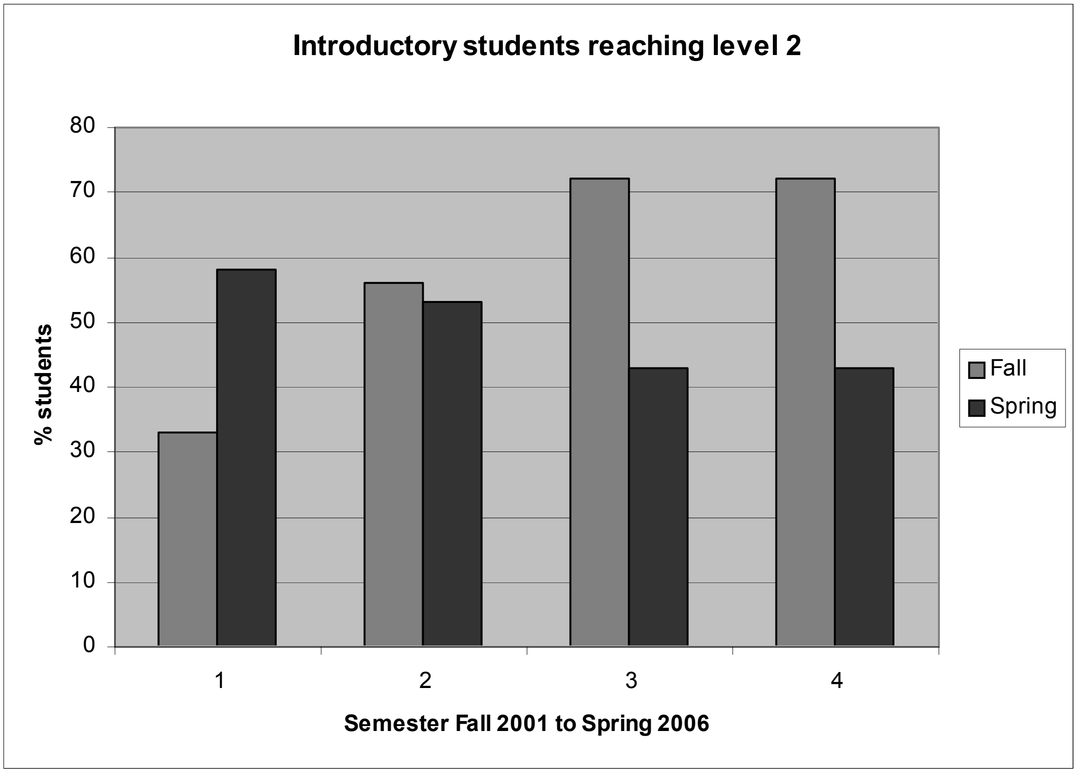

Figure 1 - Introductory students successfully reaching level 2 (note, no data is given for Fall 2002, Spring 2003 due to sabbatical leave)

of the class are older students who work full time and have more computer expertise, and second, the class is taught in one 2 hour and 45 minute block instead of two 75-mintue blocks.

Assessing the Course

My university employs an end-of-the-semester student evaluation of the course. Like surveys given at many universities the questions are not course specific and the results are not available until several weeks after the semester is over. The numeric portion of the survey is a reasonable indicator of student satisfaction (with the instructor) and the written portion of the survey, if completed, can be revealing about what the students are especially pleased or displeased with. Figure 2 presents semester evaluations for Fall 2001 through Spring 2006. In addition to the university sponsored end-of-term survey I ask the students to complete an anonymous mid-semester survey to gauge their reaction to course material, pace, assignments, and my teaching style.

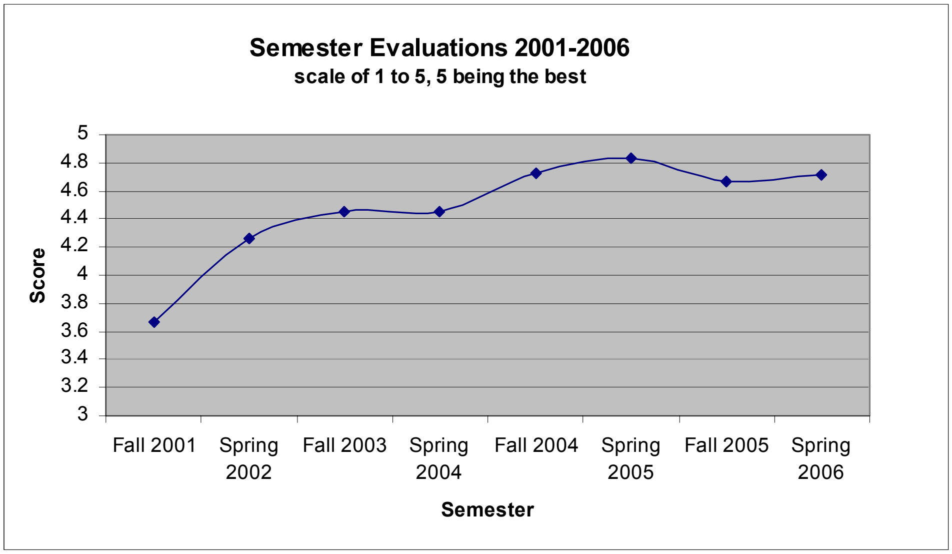

Figure 2 – Instructor Course Evaluations for Introduction to Programming (note, no data is given for Fall 2002, Spring 2003 due to sabbatical leave)

An interesting gauge of my students learning ability can be found by examining the assignments given in each semester. My confidence in the abilities of the students grew as the course evolved and improved. This raised my expectations of students in subsequent semesters. The increase in expectations and performance can be observed by gauging the function points (Pressman, 2005)

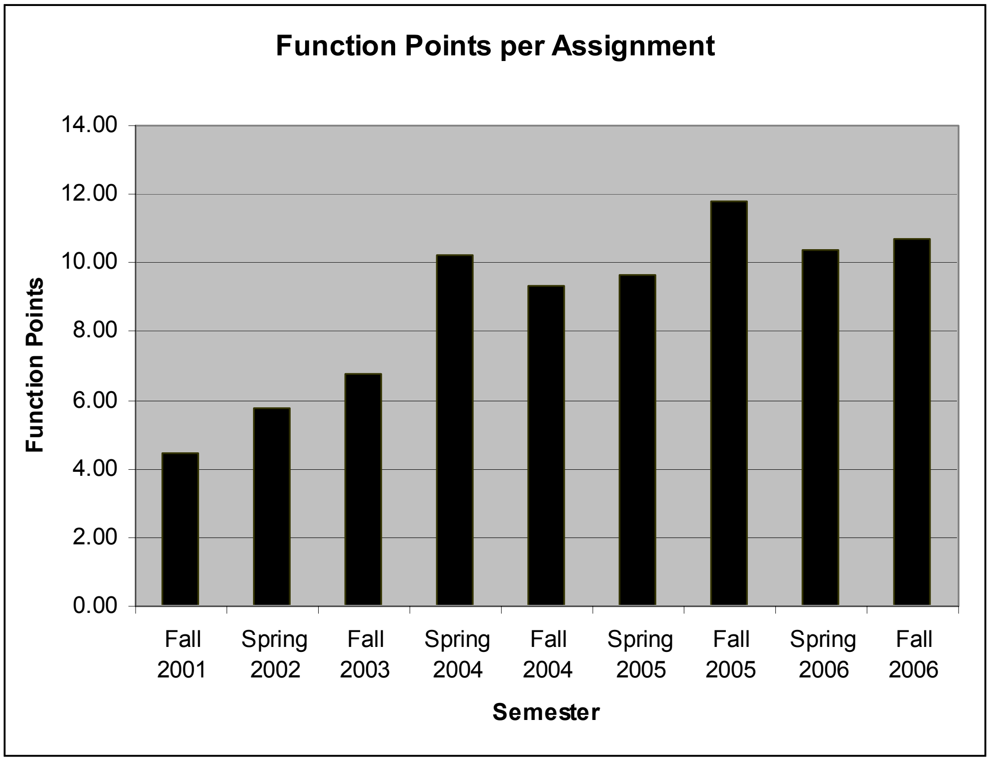

Figure 3 – Function points per assignment

per assignment. In each semester for the Introduction to Business Programming course I give my students between 8 and 10 programming assignments. A review of the last eight semesters' work shows an increase in the average number of function points per assignment from 5 in 2001 to 11 in 2006 (reference Figure 3). The function points shown are unabridged counts of the number of inputs, outputs, inquiries, and interfaces to other systems, as well as external data files.

In the spring of 2006 our university instituted the evaluation of course and curriculum via Academic Learning Compacts (ALC) (Florida Gulf Coast University, 2006). The ALCs identify the core student learning outcomes for each major. These learning outcomes are linked to the requirements of the major and the mission and goals of the college and university. Information is collected and evaluated each year to assess the effectiveness of the curriculum. The information collected from key courses includes assignments, grading rubrics, course grades, examinations, grade distributions, student self-assessments, ETS exam scores, and student surveys. The resulting report is then returned to the faculty to help improve their courses. It is too early to tell if these reports will have enough detail to be helpful to faculty. But they do constitute a form of peer review of teaching.

Conclusions

Teaching success can be measured in many ways. Deans and department heads look at enrollment rates, retention rates, and graduation rates. From their perspective, my results for the past several years are mixed. The dropout rate in introductory programming has fallen from 17% (Fall 2001) to 7% (Spring 2006), reference Figure 4, The percentage of students passing the course with an A, B, or C grade has risen from 50% in Fall 2001 to 71% in Spring 2006. Student satisfaction with the course, as measured by semester end evaluations, has risen from 3.67/5.0 (Fall 2001) to 4.71/5.0 (Spring 2006). However, enrollment in introduction to programming has de-

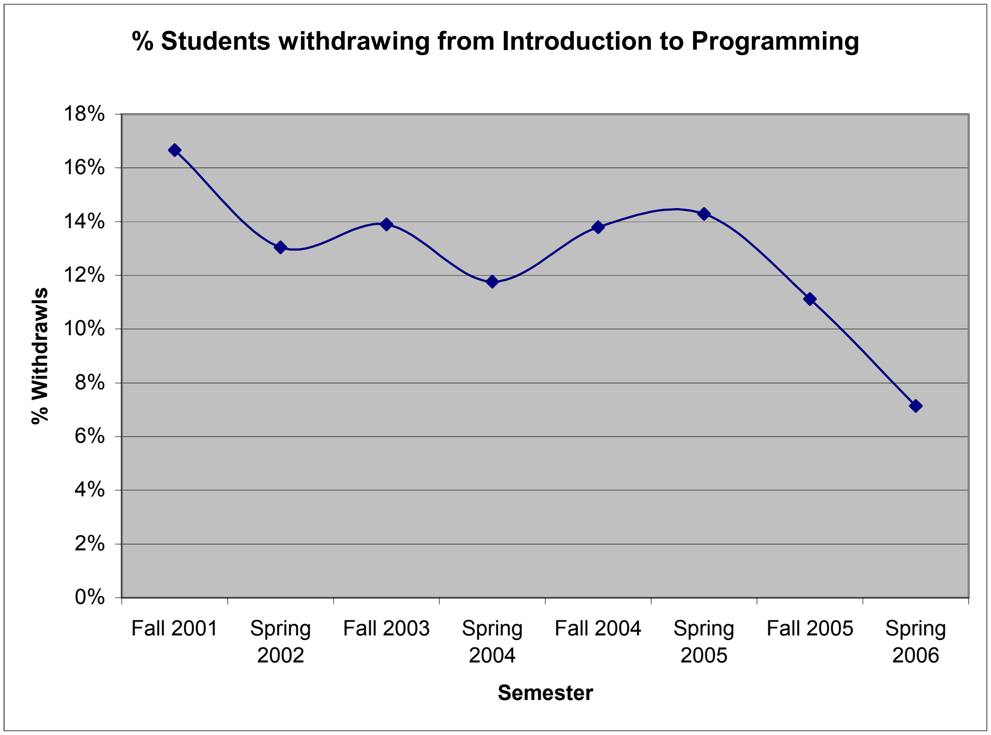

Figure 4 – Dropout rate from Fall 2001 to Spring 2006 (note, no data is given for Fall 2002, Spring 2003 due to sabbatical leave)

clined from 60 students in the 2001-2002 school year to 33 students in the 2005-2006 school year. Part of this decline can be explained by the growth of the Computer Science major that was established in fall of 2002. Enrollments for fall 2006 show a significant (40%+) increase over fall 2005.

In the Introduction to Programming class, students are successful at completing the series of short assignments, each assignment demonstrating their knowledge about one aspect of the Java language (e.g. arrays and for loops). The last few assignments in this class build upon one another to form a complete application (e.g. contact list manager). Some students do struggle initially with the Netbeans IDE, especially in regard to creating and managing projects. Part of these problems stem from previous experience with computer programs (MS Word, Excel, PowerPoint) that have only one file (.doc, .xls, .ppt) associated with an assignment. The other problem comes from their learning to move their projects to and from the computer lab to their home PCs. This problem has largely been alleviated by the use of USB memory drives (thumb drives) and/or students choosing to bring laptops to class.

In the Intermediate Programming course students are given several weeks to complete their assignments with adequate instructor contact time to solve problems. However, many students fail at completing the assignments on time because they put them off to the last week, working instead on other courses or leisure activities. Another tendency of my students is to try to write the entire program before debugging any of it. This occurs despite my best efforts to teach the incremental programming techniques. They are then faced with a program several hundred lines long and no real idea of what portion works and what does not. To combat this I have instituted a series of intermediate due dates. This forces the students to begin the projects and to do incremental programming tasks. This approach could be combined with a simple scheduling (like PERT/CPM) chart in order to give a real-world feel to the projects.

The most effective changes I've made involve setting student expectations, recognizing and addressing aspects of the Java language that are difficult to learn, incorporating a full featured IDE that has a form designer, and the use of active learning. The use of active learning gives me the opportunity to gauge student understanding and allows them to learn from one another (and one another's mistakes). Setting proper expectations helps to keep students focused on what they are to learn. Students who come into the course fearful of learning to program are less apprehensive, and those students coming into the course expecting to learn everything at once (or in one semester) are more satisfied. In the future I plan to develop a student self-assessment tool to help them gauge their progress in advancing through the stages of becoming a programmer.

References

Ala-Mutka, K., Uimonen, T., & Järvinen, H. (2004). Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, *3*(1), 245-262. Available at http://jite.org/documents/Vol3/v3p245-262-135.pdf

Association for Information Systems (2002). IS 2002 standards. Last retrieved: 27-11-2006 from http://www.aisnet.org/Curriculum/IS2002-12-31.pdf

Astrachan, O. (1998). Concrete teaching: Hooks and props as instructional technology. *ACM SIGCSE Bulletin, 30*(3), 21-24. [Proceedings of the 6th Annual Conference on the Teaching of Computing/3rd Annual Conference on Integrating Technology into Computer Science Education].

Austin, C. (2004). J2SE 5.0 in a nutshell. Last retrieved: 27-11-2006 from http://java.sun.com/developer/technicalArticles/releases/j2se15/

Bergin, J, & Naps, T. (1998). Java resources for computer science instruction. *The Working Group Reports Of The 3rd Annual SIGCSE/SIGCUE Iticse Conference On Integrating Technology Into Computer Science Education,* December, 14-34.

Binstock, A. (2006, September 18). Genuitec brightens up Eclipse. *InfoWorld*. Last retrieved: 27-11-2006 from http://www.javaworld.com/javaworld/jw-09-2006/jw-0918-iw-myeclipse.html

Bruce, C., Buckingham, L, Hynd, J.,McMahon, C., Roggenkamp, M., & Stoodley, I. (2004). Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education*, *3*(1), 143-160. Available at http://jite.org/documents/Vol3/v3p143-160-121.pdf

Culwin, F. (1999). Object Imperatives! *ACM SIGCSE Bulletin, 31*(1), 31-36. [The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education].

Dawson, R.J., & Newman, I.A. (2002). Empowerment in IT education. *Journal of Information Technology Education*, *1*(2), 125-141. Available at http://jite.org/documents/Vol1/v1n2p125-142.pdf

Decker, R., & Hirshfield, S. (1994). The top10 reasons why object-oriented programming can't be taught in CS 1. *ACM SIGCSE Bulletin, 26*(1), 51-55. [Selected Papers of the Twenty-Fifth Annual SIGCSE Symposium on Computer Science Education].

Falkman, D. (2005). IDE wars: Has NetBeans 4.1 eclipsed Eclipse? *Java Boutique*. Last retrieved: 27-11-2006 from http://javaboutique.internet.com/reviews/netbeans41/

Florida Gulf Coast University (2006). Academic learning compact for B.S. in computer information systems. Last retrieved: 27-11-2006 from http://www.fgcu.edu/OCI/70.asp

Gallardo, D. (2004). Migrating to Eclipse: A developer's guide to evaluating Eclipse vs Netbeans. IBM. Last retrieved: 27-11-2006 from http://www-128.ibm.com/developerworks/webservices/library/os-ecnbeans/

Hensel, M. (1998). The approaching crisis in computing education: Enrollment, technology, curricula, standards, and their impact on educational institutions. *Journal of Information Systems Education*, *9*(4), 24-27.

Kölling, M., Koch, B., & Rosenberg, J. (1995). Requirements for a first year object-oriented teaching language. *ACM SIGCSE Bulletin, 27*(1). [Papers of the 26th SIGCSE technical symposium on Computer science education].

Lewis, J.P., & Neuman, U. (2003). Performance of Java versus C++. Last retrieved: 27-11-2006 from http://www.idiom.com/~zilla/Computer/javaCbenchmark.html

Liang, Y.D., (2007). *Introduction to Java programming: Comprehensive version* (6th ed.). Prentice Hall.

Mangione, C., (1998). Performance tests show Java as fast as C++. *JavaWorld*, Last retrieved: 27-11-2006 from http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf_p.html

McConnell, J., (1996). Active learning and its use in computer science. *ACM SIGCSE Bulletin, 28*(1), 52-54. [Proceedings of the Conference on Integrating Technology into Computer Science Education].

Naugler, D., & Surendran, K. (2004). Simplicity first: Use of tools in undergraduate computer science and information systems teaching. *Information Systems Education Journal, 2*(5), 3-11.

Noll, C. L., & Wilkins, M. (2002). Critical skills of IS professionals: A model for curriculum development. *Journal of Information Technology*, *1*(3), 143-154.

Pendergast, M. (2005). Teaching Java to IS students: Top ten most heinous programming errors. *Proceedings of the Association of Information Systems Americas Conference AMCIS 2003*, Omaha, NE, August 2005

Pressman, R.S. (2005). *Software engineering: A practitioner's approach* (6th ed). McGraw-Hill.

Reddy, A. (2000). Java coding style guide. *Sun MicroSystems*. Last retrieved: 27-11-2006 from http://developers.sun.com/prodtech/cc/products/archive/whitepapers/java-style.pdf

Sim, E. & Wright, G. (2002). A comparison of adaptation-innovation styles between information systems majors and computer science majors. *Journal of Information Systems Education*, *13*(1), 29-35.

Sun MicroSystems. (2004). The Mars mission continues. Last retrieved: 27-11-2006 from http://www.sun.com/aboutsun/media/features/mars.html

TIOBE Software (2006). TIOBE programming community index for November 2006. Last retrieved: 27-11-2006 from http://www.tiobe.com/index.htm?tiobe_index

Tyma, P. (1998). Why we are using Java, again? *Communications of the ACM*, *41*(6), 38-42.

Udell, J. (2002, July 5). Eclipse casts shadows. *InfoWorld.* Last retrieved: 27-11-2006 from http://www.javaworld.com/javaworld/jw-07-2002/jw-0705-iw-eclipse.html

White, G., & Sivitanides, M. (2002). A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics. *Journal of Information Systems Education, 13*(1), 60-66.

Zweben, S., & Aspray. W. (2004). Undergraduate enrollments drop; Department growth expectations moderate. *2002-2003 Taulbee Survey*, Computing Research Association. Last retrieved: -27-11-2006 from http://www.cra.org/CRN/articles/may04/taulbee.html

Biography

Dr. Pendergast is an Associate Professor in the Computer Information Systems Department within the College of Business at Florida Gulf Coast University. Dr. Pendergast has a M.S. and Ph.D. in Management Information Systems from the University of Arizona and B.S.E. in Electrical Computer Engineering from the University of Michigan. He has worked as an analyst and engineer for Control Data Corporation, Harris Corporation, Ventana Corporation, and as an Assistant Professor at the University of Florida and Washington State University. His research interests include computer-supported cooperative work, data communications, software engineering, terrain modeling, and group support systems.