T.C.

BAHÇEŞEHİR ÜNİVERSİTESİ

INVESTIGATING LEARNING METHODOLOGIES OF OBJECT ORIENTED PROGRAMMING

Master's Thesis

Duygu ÇAKIR

İstanbul, 2010

T.C.

BAHÇEŞEHİR ÜNİVERSİTESİ
The Graduate School of Natural and Applied Sciences
Computer Engineering

INVESTIGATING LEARNING METHODOLOGIES OF OBJECT ORIENTED PROGRAMMING

Master's Thesis

Duygu ÇAKIR

ADVISOR: Assoc. Prof. Adem KARAHOCA

İstanbul, 2010

T.C.

BAHÇEŞEHİR ÜNİVERSİTESİ
The Graduate School of Natural and Applied Sciences
Computer Engineering

Title of Thesis: Investigating Learning Methodologies of Object Oriented Programming
Name/Last Name of the Student: Duygu ÇAKIR
Date of Thesis Defense: September 13, 2010

The thesis has been approved by the Graduate School of Natural and Applied Sciences.

Asst. Prof. Dr. Tunç BOZBURA
Director

This is to certify that we have read this thesis and that we find it fully adequate in scope, quality and content, as a thesis for the degree of Master of Science.

Examining Committee Members:	Signature
Assoc. Prof. Dr. Adem KARAHOCA:	_____
Asst. Prof. Dr. Alper TUNGA:	_____
Asst. Prof. Dr. Yalçın ÇEKİÇ:	_____

To my loved ones…

ACKNOWLEDGEMENTS

I am thankful to my advisor Assoc. Dr. Adem Karahoca for all of his support, insight, and invaluable help during the preparation and information collection for this thesis; and, in general, for being the person who made it possible the beginning of my master studies at Bahçesehir University. I am really grateful to him for his great support, his enthusiasm with my study and his unconditional trust.

I am also grateful to my dear teachers, Asst. Prof. Orhan Gökçöl and Selvihan Nazlı Kaptan, for taking me as their teaching assistant and encouraging me through my academic studies.

My special thanks go to my friends and colleagues, especially to Çağrı Özgün, for their endless support all through this hard work and also my personal life.

I would really like to thank to my beloved fiancée and my family for their understanding, the opportunities they offered, and belief in me. Their acknowledgement was the source of my strength and determination in life.

ABSTRACT

INVESTIGATING LEARNING METHODOLOGIES OF OBJECT ORIENTED PROGRAMMING

Çakır, Duygu

Computer Engineering

Thesis Advisor: Assoc. Prof. Adem Karahoca

September 2010, 56 pages

In this study, the main objective is to analyze object oriented learning methodologies by examining the objects learned by the students. During the last three years, object oriented programming language teaching is evaluated and assesed by using exams, quizes, homeworks and finals. Evaluation data were collected and analyzed. The main purpose of the study is to improve object-oriented programming syllabi and increase the achievements of the students of the course. The students' ability of self improvement will be assessed by making some inventories. Some of the main topics included in the syllabus are as follows: class structure, constructors, functions and prototypes, declaring and initializing instances, abstraction and encapsulation.

Keywords: Object Oriented Programming, Syllabus Design, Software Engineering

ÖZET

Çakır, Duygu

Bilgisayar Mühendisliği

Tez Danışmanı: Doç. Dr. Adem Karahoca

Eylül 2010, 56 sayfa

Bu çalışmadaki temel amaç, öğrencilere öğretilen konuları inceleyerek nesneye dayalı programlama metotlarını analiz etmektir. Geçtiğimiz üç sene boyunca yapılan nesneye dayalı programlama derslerinin sınavları, quizleri, ödev ve finalleri toplanmış ve değerlendirmeye uygun bulunmuştur. Çalışmanın temel hedefi, nesneye dayalı programlama dersinin müfredatını geliştirmek ve öğrencilerin derse yönelik kazanımlarını artırmaktır. Öğrencilerin kişisel gelişimleri, bu ölçümlerin sonucuna göre değerlendirilecektir. Müfredatta ele alınan ana başlıklardan bazıları şunlardır: sınıf yapısı, yapılandırıcılar, fonksiyonlar ve prototipler, değişken yaratma ve değer atama, soyutlama ve kapsülleme.

Anahtar Kelimeler: Nesneye Dayalı Programlama, Müfredat Tasarımı, Yazılım Mühendisliği

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Object Oriented Programming	: OOP
Object Oriented Programming Languages	: OOPL
Computer Engineering	: CE
Software Engineering	: SE
Industrial Engineering	: IE
Electrical and Electronical Engineering	: EEE
Mechatronics Engineering	: ME
Environmental Engineering	: EE
Java Virtual Machine	: JVM
Java Server Pages	: JSP
Hyper Text Markup Language	: HTML
World Wide Web	: WWW
Software Development Kit	: SDK
Integrated Development Environment	: IDE
Statistical Package for Social Sciences	: SPSS
University Entrance Exam for Students	: ÖSS

1. INTRODUCTION

1.1. BACKROUND

Object oriented programming (OOP) has been in our lives for more than 30 years now. Since the day it was born, it brought a new perspective to programming, especially in universities and high schools where programming courses are given.

As in many universities abroad, Bahçeşehir University gives CS1 & CS2 (Introduction to Programming & Object Oriented Programming) courses to all departments of the Engineering Faculty and also to some departments of the Arts and Sciences Faculty.

1.2. PROGRAMMING LANGUAGE COURSES

Bahçeşehir University has been teaching programming languages in the first year of Computer Engineering (CE), Software Engineering (SE), Industrial Engineering (IE), Electrical and Electronics Engineering (EEE), Mechatronics Engineering (ME), and Environmental Engineering (EE).

Our CE department teaches this first year course using C++, a hybrid language which derives from C. The syntax is slightly difficult considering the other languages like Java, VB, C#, etc…

Our SE department is a newer department (6 years old) in comparison to the CE department (12 years). The SE department has Java as the object oriented programming language in the introduction to programming and object oriented programming courses.

The other departments of the Engineering Faculty also teach programming language courses but they do not continue these language courses as the department itself, unless the student wants to get it from other departments as an outsider.

IE and EE teach VB, EEE and ME teaches C to the freshmen during programming language courses.

1.3. THESIS OUTLINE

During this thesis, a general outline will be drawn to the problem by first mentioning the relevant background of OOP.

“Introduction to Programming with Java” and “Object Oriented Programming with Java” is given to the SE freshmen. During the first year, we teach the basics of Java and OOP and its applications respectively in two semesters. Eclipse IDE (Classic) is used as the Java editor (eclipse.org).

In this study, Java students’ quizzes, midterms, finals and homeworks were collected between the years 2007 and 2010. These data were analyzed to obtain the success results of students to show the advantages of the learning methodologies.

The data collected are collected from the SE programming courses (given in Java) which are given to three different types of departments: CE, SE and Math & Computer Sciences freshmen.

2. REVIEW OF PROGRAMMING LANGUAGES

Pears, et. al. (2007), discussed the improvement of OOP course and came up with these questions:

- "What programming language should be used?" – Language Choice
- "What tools and environments support learning, and how?" – Tools for Teaching
- "What pedagogies have been tested, and what are the outcomes?" - Pedagogy
- "How does the new course fit into a larger computing curriculum?" – Syllabus

Thimbleby (2003) agrees and outlines a number of desirable properties that a literate program should exhibit, such as;

- Documentation and code should develop together and be tightly coupled;
- Editing must be possible without affecting the integrity of the documentation and code;
- Tool support must be lightweight, easy to use, and discourage manual "touch ups";
- The tool must scale;
- Fragments of code should be explainable in any order;
- Readers of the documentation should not have to face special notation or conventions;
- The tool should be language independent;
- Any required translations from documentation to code should be automated; and
- The tools must be simple.

The aim, under this topic, is to find an answer to each question. Let us start with the question "Why Object Oriented?" and then move on to why we chose Java and which tool we use to write our programs.

2.1. WHY OOP?

"Object-oriented languages allow the building of software from parts, encouraging code reuse and encapsulation through the mechanisms of inheritance and polymorphism. Commonly, Object-oriented languages also allow dynamic binding of method calls, dynamic loading of new classes, and querying of program semantics at runtime using reflection" (Ryder, 2003).

Wegner characterizes object oriented programming languages (OOPL) as follows: "object-oriented = objects + classes + inheritance" (Wegner, 1987). In this definition, language features such as object, class, and inheritance are emphasized. Other examples of OOPL features can also be given, such as encapsulation, operator overloading, garbage collection, metadata, etc.

Müller (1993) pointed to the three important features of OOP as encapsulation, inheritance, and message passing.

Encapsulation:

> Because it is not allowed for unattached procedures to manipulate an object the data is encapsulated. Therefore the effects of changing data and / or procedures are always restricted and easy to localize.

Inheritance:

> Inheritance is a technique which enables the reuse of behavior of already defined classes in the definition of a new class. Inheritance helps to avoid the duplication of code and the need for coding from scratch.

Message Passing:

> A message is sent to an object and represents a request to perform some action. It is in the responsibility of the receiver how to react. Thus, it is possible that different object react differently after receiving the same message.

Snyder (1986) describes the characteristics and steps of OOP methodology as follows:

- designers define new classes (or types) of objects
- objects have operations defined on them
- invocations operate on multiple types of objects (i.e., operations are generic)
- class definitions share common components using inheritance

Pokkunuri describes "object" as an autonomous entity with its private data and methods. Its behavior is characterized by the actions that it suffers and that it requires of other objects. Data being private to the object, the important responsibility of selecting the compatible is thrust upon the object – the supplier of the service. This contrasts the style

of conventional programming wherein the consumer of the service has to select compatible operator required for the data on the hand (Pokkunuri, 1992).

Kristensen (1996) pointed out that an OOPL should support the following ideas:

- **Enforce**: The language forces tile programmer to use tile concept.
- **Encourage**: The language provides convenient mechanisms to express the concept, but the programmer can choose not to do so.
- **Enable**: The language does not have any language mechanisms to express the concept, but the programmer can easily establish a convention of using other mechanisms for tile same purpose.
- **Discourage**: The language does not contain any language mechanisms to support the concept, and it will take extraordinary skills or great discipline to establish a simulation of the concept.
- **Prohibit**: The language semantics is such that any attempt to use the concept are hindered by the language.

Unlike Kristensen, Arif (2000) points out that "object-oriented programming principles and concepts could be easily simplified and taught to the students in this course". He also agrees with Wegner, and also highlights the importance of classes, objects, inheritance and adds polymorphism to these important features. According to his studies, he states that for students, accepting the idea of classes and objects seems to be the most important yet difficult one. He finds the solution as moving the students' programming behavior from structural programming towards object-oriented programming.

According to Arif's survey on students, 94% of his students chose OOP to be their preferable programming approach in the future, rather than a non-object-supportive language. 97% of his students think that OOP approach helps them in organizing their programs.

There exist 4 types of OOPLs:

- Pure OOPLs – Where everything is considered and treated as an object, e.g. Smalltalk, Eiffel, Ruby, etc.
- OOPLs with some procedural Elements – Not everything is treated as an object, there exists some free procedural elements, e.g. C++, C#, Java, etc.
- OOPLs which were previously procedural languages – e.g. Fortran, Perl, PHP, etc.

- Object-Based Languages – In order to create an object, one does not need to create a class first; objects are just collection of methods, e.g. Self, ECMAScript, JavaScript, JScript, ActionScript, etc.

2.2. THE PEDAGOGY

Computer science (CS) instructors look for "good" examples that allow in-depth discussion of the fundamental concepts of object-oriented programming (OOP), yet keep the implementation framework simple (Ragonis, 2010).

A debate is taking place in many departments of computer / information science about the best way to approach the teaching of programming. Should student be exposed immediately to the new paradigm of OOP, using a language like C++ or Java, or should they be taught with a more gradual approach? (Burton, et. al., 2003)

2.2.1. MATHEMATICAL BACKGROUND

More than 80% of the students in their first year start the university as soon as they finish high school. This is a huge fact that their mathematical intelligence is still fresh. We believe that the minute they step their feet to the university, we should take advantage of their fresh math knowledge and use it in advance of our goal on teaching object oriented programming. Using their intelligence, we try to teach them how to solve problems in a systematic way and then gradually make them memorize new keywords and rules. This is called the "Inductive Procedural Approach" to OOP (Cakir, et. al., 2010).

Instead of choking the student into new rules and syntax, as in the objects-first methodology, we argue that the student masters on the algorithms first, (s)he can write a more effective and costless code. Lewis (2000) agrees with us on this in a more moderate way: "object-first methodology is not a good pedagogy for teaching object orientation".

Cecchi (2003) agrees with us: "We believe that the two paradigms are not mutually exclusive: indeed, when designing a complex object system, the first phase of creating the objects' relationships is necessarily followed by an implementation phase, which requires a good knowledge of structured programming." Hu (2004) has bigger concerns on the object-first methodology: "One important reason for educator to argue against the

objects early approach is their concern that starting with object results in the learning of algorithmic problem-solving to be neglected."

Burton and Bruhn(2003) finalize the problem: "is almost impossible to separate programming issues completely from mathematical issues … Mathematics is the language of science, including computer science and information science … The authors believe that if the teaching of programming is approached in a gradual and structured way, and if the students enter a programming course with the right kind of mathematical skills, then the negative perception of programming as being too difficult can be mitigated".

2.2.2. CODING STANDARDS

The pedagogical issues of the student don't always come up according to their understanding of the language and methodology but also considers the industry too. The selection of the programming language should consider pedagogical issues, such as using a language that is simple, that supports a given paradigm (either procedural or object oriented) and, not to be underestimated, that satisfies pragmatic goals of value to industry (Cecchi, et. al., 2003).

Our other concern was to give the student "the unwritten coding rules", such as the indentation, the discipline, the naming, and so on. To change one's habit is hard, but to make him gain a new habit is much easier. That's why we chose to start from the beginning in an inductive way.

Roy (2006) mentioned the advantages of a good written and organized code as follows:

- More efficient algorithm development
- Reduction on coding errors
- More readable code, especially for non-author readers
- Better management of complex systems for integration, maintenance and support

Using these rules and concepts, Deimel and Neveda gave some tips on creating code for better readability. Some of these tips we should pay attention are;

- Inconsistencies between comments and codes
- Use of indentation to show structure

- Use of step-wise abstraction
- Use of program slicing to isolate behavioral components.

According to Li and Prasad (2005), programming and software development courses are the most suitable courses for generating coding skills. "Complying with given coding standards is a vital professional skill required by the software industry; one that ought to be actively developed within IT education" they say. Our aim is to justify their opinion and transfer the coding discipline to the students.

2.2.3. PLANNING

Most texts used to teach beginners to program focus on presenting language constructs, programming language concepts, and computer programs (complete or partial) (Caspersen, 2006).

After pointing out the problem, Caspersen offered two solutions to the problem:

- Teach students about the process of software development, to enable them to follow organized steps to move toward a solution to a problem, and
- Treat software development explicitly as a process that is carried out in stages and small steps, rather than the writing of a single, monolithic solution.

Corresponding with the solutions that Caspersen offered, before starting to write codes, we first teach how to write a pseudo code to a problem and design a flowchart to the code.

A cognitively complex activity such as programming cannot be done entirely in the head, but must be supported by external aids that redistribute the cognitive complexity, thus allowing programmers to produce better solutions and to tackle more complex programming problems. To complement the use of programming languages and programming environments, programmers make use of alternative notations such as pseudo code.

Pseudo code is the term usually used to refer to informal textual representations of a program or algorithm (Bellamy, 1994).

Pseudo code aims to fill the gap between the informal (spoken or written) description of the programming task and the final program (code) that can be executed, or at least automatically converted into an executable form (Roy, 2006).

Pseudo code generally includes the following:

- The use of English-like statements to describe the computational task and/or process
- Some reserved words or symbols (nouns and verbs) to describe common processes and actions
- Ways to describe standard computational tasks

The advantages of writing a pseudo code to the problem can be summarized to the topics below (Roy, 2006):

- Pseudo code provides an effective vehicle for describing computational processes.
- A combination of text and graphics allow the pseudo code to be defined, edited, and when compared to raw code, displayed with increased readability.
- Though a process of stepwise refinement the actual code can be built progressively, until the complete program is fully specified and operational.
- By tightly coupling the pseudo code with comments and code segments, the resulting code can reach the level of literate programming.

Scanlin (1988) found out that the graphical flowchart provided a clear benefit to the student reader because textual pseudo code is mainly processed by the brain's left hemisphere (verbal, logical, sequential), while the flowchart can also effectively utilize the right hemisphere (visual, , simultaneous) at the same time. The flowchart, with both text and graphic notation, can thus make more effective use of brainpower.

Cross and Sheppard (1988) worked on a variety of graphical pseudo code representations, including ANSI flowcharts (which we used as we taught), Action Diagrams (Martin, et. al., 1985), Control Structure Diagrams (Cross, 1986), Nassi – Shneiderman diagrams (Nassi, et. al., 1973), and Warner – Orr Diagrams (Orr, 1977). Their entire goal is to provide a clear picture of the structure and semantics of the program through a combination of graphical constructions and some textual annotations.

Caspersen (2006) wrote the pseudo code of OOP:

Step 1: Create the class

> The first step towards implementation is to create an implementation class that provides methods with the intended signatures. The method implementations at this stage are stubs (i.e. minimal method bodies).
>
> For methods that do not return values, the method body is empty.

For methods with return values, the method body consists of a single return statement. The value returned is a default value (zero for numbers, null for object types, etc.)

Step 2: Create tests

Once method stubs have been defined, test cases can be written for every method.

Step 3: Alternative representations

For instance, convert a for loop into a while loop, and then to a do-while loop, and then make it a recursive function.

The idea is making a transition from easy ➔ hard ➔ trivial ➔ challenging.

Step 4: Instance fields

When the programmer settles on one particular representation, he can refine his implementation class, and define the fields needed to represent the object.

Step 5: Method implementation

While there still exists an incomplete method in the class body, the programmer should implement, test, and finish the method.

Burton and Bruhn (2003) wrote a pseudo code for procedural programming in detail, recommending to write a pseudo code before starting to write the original code:

1) Read and understand the problem

2) Devise a solution to the problem

3) Formalize the solution as an algortihm, that is, as a sequence of steps that can be automated

4) Write the program

5) Test and debug the program

6) Document the program

Grissom (2004) simplified these steps and called them the "Software Development Lifecycle":

1) Identify specifications

2) Design a solution

3) Implement the code

4) Test

Using a pseudo code, the programmer, i.e. the student, can represent processes of sequence, iteration, selection, recursion, user input, output, file operations, function and method definitions to modularize the program and to allow the reuse of the code and for information hiding, and so on.

Hamilton and Haywood's (2004) survey results imply that conception is more important than experince, and that's why it's more important that the student learns how to find a solution to a problem using a pseudo code than directly starting to write the program itself: "The results indicate that prior programming experience is not necessary for a student's success in a course that expects them to undertake analysis and design activities for a large-scale software product."

Caspersen and Kölling (2006) gave some additional rules on how to write the simples code by adding the modularity of OOP to consideration.

Special Case rule:

> If you write code to treat a special case in your algorithm, treat the special case in a separate method.

Nested Loop rule:

> If you have a nested loop, move the inner loop into a separate method.

Code Duplication rule:

If you write the same code segment twice, move the segment into a separate method.

Hard Problem rule:

> If you need the answer to a problem that you cannot immediately solve, make it a separate method.

Heavy Functionality rule:

> If a sequence of statements or an expression becomes long or complicated, move some of it into a separate method.

Software development is a process that has to be carried out and completed in organized, small steps moving towards the solution rather than writing a single block of solution.

In 2005, Olsen summarized the importance of writing a pseudo code and using the students' mathematical intelligence all in one: "Recommended methods for teaching students how to solve problems include a focus on mathematics, flowcharts, UML, pseudo code, and other methods"

2.3. LANGUAGE CHOICE

During the last four decades, many languages have been used for teaching introductory programming. The language choice is usually made locally, based on factors such as faculty preference, industry relevance, technical aspects of the language, and the availability of useful tools and materials. The process has become increasingly cumbersome as the number of languages has grown (Pears, et. al., 2007).

Emphasizing object-orientation is an increasingly common approach taken by many computer science educators in CS1 and CS2. The two most important aspects of object-orientation are inheritance and polymorphism. Programming projects assigned to students must be designed in reasonable size but this hinders the student's experiencing developing relatively large applications (Grissom, 2004).

Schneider (1978) mentions that the targets of an introductory programming course should be problem solving and algorithm development. According to Schneider, a programming language "should be based on two critical and apparently opposing criteria: richness and simplicity – rich in those constructs needed for introducing fundamental concepts in computer programming (but) simple enough to be presented and grasped in a one semester course." If a language has a small set of constructs/features, although it may be possible to say what one wants, it may be excessively complicated to do so. On the other hand, if the language has a very large set of constructs/features, it may be difficult to assimilate them all.

Mannila and deRaadt (2006) made a census on the selection of programming languages. The most prominent reason turned out to be the industry relevance, even before pedagogical reasons. The results are shown in Table 1 and the distribution can be seen on Figure 1.

Table 1 – Language Choice Reasons

Reason	Count
Industry Relevance / Marketable / Student demand	33
Pedagogical benefits of language	19
Structure of degree / Department politics	16
OOP language wanted	15
GUI interface	6
Availability / Cost to students	5
Easy to find appropriate texts	2

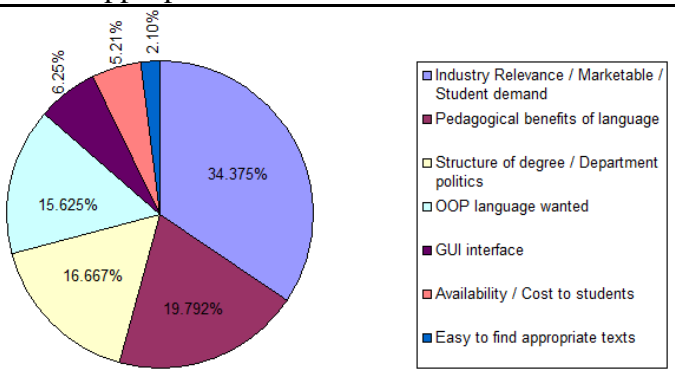

Figure 1 – Language Choice Reason Distributions

Over the last twenty years, the choice of which language should be the most appropriate to be taught in courses like CS1 has shifted from procedural languages such as Pascal or C, to object oriented languages such as C++ and Java (Cecchi, 2003).

Mannila and deRaadt (2006) made a survey on languages compared by their features (Table 2).

Table 2 – Languages Compared by Features

		C	C++	Eiffel	Java	JavaScript	VB
Learning	Is suitable for teaching			1			
	Can be used to apply physical analogies			1	1	1	1
	Offers a general framework	1	1	1	1	1	1
	Promotes a design driven approach for teaching software			1	1		
Design and Environment	Is interactive and facilitates rapid code development						
	Promotes writing correct programs		1	1	1		
	Allows problems to be solved in "bite-sized chunks"	1	1	1	1	1	1
	Provides a seemless development environment				1		
Support and Availability	Has a supportive user community	1	1	1	1	1	1
	Is open source				1		
	Is consistently supported across environments	1	1	1	1	1	
	Is freely and easily available	1	1	1	1	1	
	Is supported with good teaching material		1	1	1		1
Beyond Introductory Programming	Is not only used in education	1	1	1	1	1	1
	Is extensible	1	1	1	1		1
	Is reliable and efficient	1	1	1	1	1	1
	Is not an example of the QWERTY phenomena		1	1	1	1	1
	TOTAL POINTS:	**8**	**11**	**14**	**15**	**9**	**9**

As can be seen from the table above, which is a slightly new poll since it was made in 2006, Java has the biggest score according to the criteria.

Today, C, Java and C++ top the list of the most widely used programming languages, both in industry and education (Pears, et. al., 2007).

The TIOBE community index tracks the community once a month and measures the programming language trends according to the search engines' ranks. September 2010 values show that Java is the most popular programming language with 17.9% usage, C and C++ follow Java with 17.15% and 9.8% respectively (Table 3). But this table's top

three hasn't changed for five years. Ten years ago, when it first came out, Java wasn't even on the ranking, but after it came out, it became the first and hasn't moved down yet (Table 4).

Table 3 - Programming Language Ranks by Sept 2010

Position Sep 2010	Position Sep 2009	Delta in Position	Programming Language	Ratings Sep 2010	Delta Sep 2009	Status
1	1	=	Java	17.915%	-1.47%	A
2	2	=	C	17.147%	+0.29%	A
3	4	↑	C++	9.812%	-0.18%	A
4	3	↓	PHP	8.370%	-1.79%	A
5	5	=	(Visual) Basic	5.797%	-3.40%	A
6	7	↑	C#	5.016%	+0.83%	A
7	8	↑	Python	4.583%	+0.65%	A
8	18	↑↑↑↑↑↑↑↑↑↑	Objective-C	3.368%	+2.78%	A
9	6	↓↓↓	Perl	2.447%	-2.08%	A
10	10	=	Ruby	1.907%	-0.47%	A
11	9	↓↓	JavaScript	1.665%	-1.33%	A
12	11	↓	Delphi	1.585%	-0.39%	A
13	13	=	Lisp	1.084%	+0.24%	A--
14	12	↓↓	Pascal	0.790%	-0.17%	A--
15	27	↑↑↑↑↑↑↑↑↑↑↑↑	Transact-SQL	0.771%	+0.40%	A--
16	-	↑↑↑↑↑↑↑↑↑↑	Go	0.728%	+0.73%	A--
17	21	↑↑↑↑	RPG (OS/400)	0.715%	+0.26%	A--
18	30	↑↑↑↑↑↑↑↑↑↑	PowerShell	0.686%	+0.42%	B
19	24	↑↑↑↑↑	Ada	0.676%	+0.29%	B
20	14	↓↓↓↓↓↓	PL/SQL	0.637%	-0.18%	A-

Table 4 - Last 5, 10, 15 years of Ranking

Programming Language	Position Sep 2010	Position Sep 2005	Position Sep 1995	Position Sep 1985
Java	1	1	-	-
C	2	2	1	1
C++	3	3	2	10
PHP	4	5	-	-
(Visual) Basic	5	6	3	4
C#	6	7	-	-
Python	7	8	21	-
Objective-C	8	44	-	-
Perl	9	4	8	-
Ruby	10	25	-	-
Lisp	13	14	7	2
Ada	19	17	6	3

The below is the figure representation of the top 10 popular languages since 2000 (Figure 2)

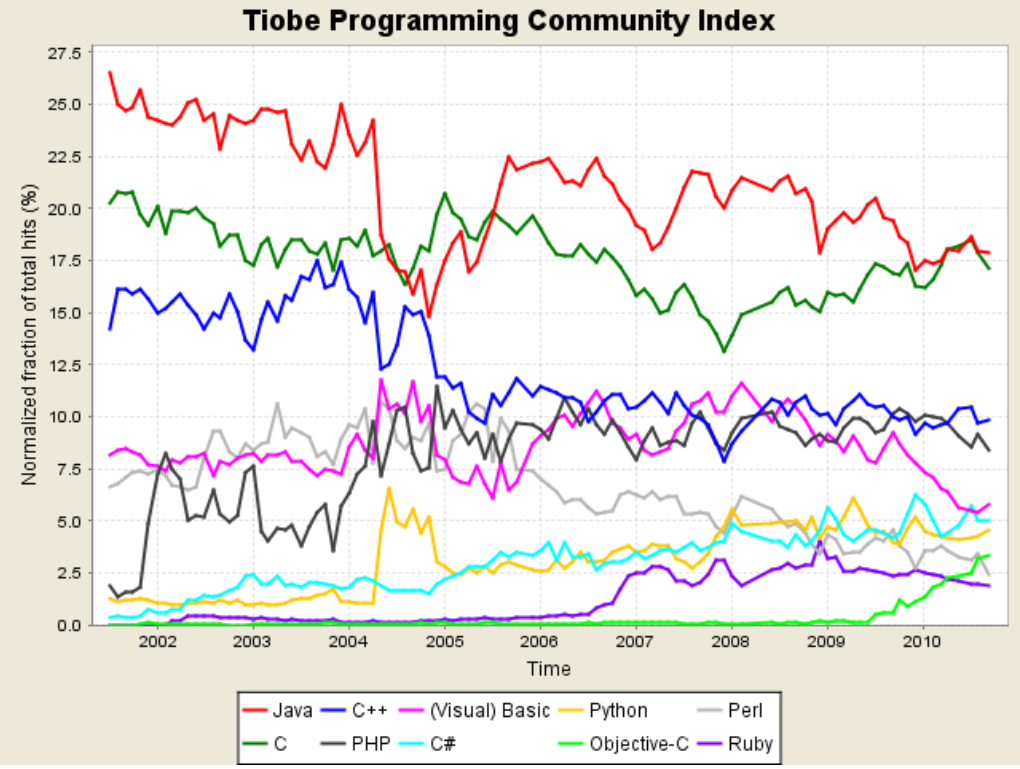

Figure 2 - Top 10 Languages' Evolution Since 2010

These charts and surveys made indicate that the most widely used languages have not changed during this time span. Java, C++, and C have continuously been among the top four.

In 1991, James Gosling lead a team at Sun Microsystems that developed the first version of Java (which was not yet called Java). This first version of the language was designed for programming home appliances, such as washing machines and television sets (Savitch, 2003).

Java (as we know) was introduced in 1995, the result of an internal research project at Sun Microsystems led by James Gosling (other key contributors include Bill Joy, Guy Steele and Gilad Bracha). The language came at just the right time to benefit from two separate phenomena (Meyer, 2009):

- Widespread dissatisfaction, after initial enthusiasm for object technology in the late eighties, with the C++ language (see appendix C), particularly its complexity and the limits of its "hybrid" approach retaining compatibility with the non-object-oriented C language.

- The spread of Internet access and the advent of the World-Wide Web, which seemed to call for a universal mechanism to execute programs securely from within browsers.

The current custom is to name programming languages according to the whims of their designers. Java is no exception. There are conflicting explanations of the origin of the name "Java." Despite these conflicting stories, one thing is clear: The word "Java" does not refer to any property or serious history of the Java language. One believable story about where the name "Java" came from is that the name was thought of when, after a fruitless meeting trying to come up with a new name for the language, the development team went out for coffee, and hence the inspiration for the name "Java." (Savitch, 2003)

Since Java has become a popular programming language and widely used by computing professionals, many schools and universities began to switch to Java as the introductory programming language. The abundance of packages that come with the Java Software Development Kit (JDK) provides a seamless transition for the students in their higher level courses. For instance, they can use the java.sql package in their database course, the Java Collection Framework for a data structures course, java.net and java.rmi for computer network and data communication courses, etc. With other programming languages, this seamless transition is difficult to achieve. For instance, C or C++ programmers who want to write GUI programs must rely on a third party library that may be supported only on specific hardware platforms (Grissom, 2004).

Sun Microsystems (1995) describe Java as: “A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language”.

Java could well be the answer to the problem of choosing an appropriate language for the first programming course. Java appears to have outstanding prospects for computer science education in general and the first programming course in particular (King, 1997).

Java is a general-purpose, object-oriented language. Most of Java’s press coverage emphasizes its client/server role, as a language for writing “applets” that are downloaded from a server and executed locally. But Java isn’t restricted to writing applets; it works just as well for writing traditional single-computer applications (King, 1997).

2.3.1. PROPERTIES OF JAVA

Java works on the virtual machine (JVM), which serves a close connection between the programming language and the computer platform. JVM is the software system that provides mechanisms to support execution of Java programs. The JVM converts the code into bytecodes, which can then be interpreted (or compiled to machine code) on any platform. This principle is called the “Write once, run anywhere” principle.

- A **class loader** manages classes and libraries in the file system and dynamically loads classes in bytecode format.
- A **verifier** checks that bytecode satisfies fundamental constraints on reliability and security: type safety (non-null references always lead to objects of the expected types); information hiding (feature access observes visibility rules); branch validity (branches should always lead to valid locations); initialization (every data element is initialized before use).
- An **interpreter**, the software equivalent of a CPU in a physical computer, executes bytecode.
- A **Just In Time compiler** (**JIT** compiler or “jitter”) translates bytecode into machine code for a specific platform, performing various optimizations. (Meyer, 2009)

Java is also a familiar language, which was derived from C – C++. The general language and syntax basis were taken from those two languages and were then adapted

to Java. It follows all the general programming features like loops, data types, conditions, curly braces, semi-colon etc. It's a fully featured OOP language as it supports all OOP features including classes, modules, inheritance, Polymorphism etc.

2.3.2. WHERE JAVA IS USED

Java is available in many different forms and places such as:

JSP: Like PHP and ASP, Java Server Pages are based on a code with normal HTML tags, which helps in creating dynamic web pages.

Java Applets: Java Applets are used within a web page to add many new features to the web browser. They are commonly used in instant messaging programs, chat services, and so on.

J2EE: The software Java 2 Enterprise Edition is used to transfer data based on XML structured documents between one another.

JavaBeans: JavaBeans is a reusable software component that can easily be assembled to create new and advanced applications (RoseIndia).

Besides these, we face with Java in our every day life. Decoders, printers, games, navigation systems, web cams, medical devices and parking machines also use Java coding (Cakir, et. al., 2010).

2.3.3. ADVANTAGES OF JAVA

Java programs, like those in other object-oriented languages, are structured into classes, but Java offers a modular structure above the class level: the package. A package is a group of classes.

Packages fulfill three main roles. The first is to help you structure your systems and libraries. Packages can be nested, and hence make it possible to organize classes in a hierarchical structure.

The second role of packages is as compilation units. Rather than compiling classes individually, you can compile an entire package into a single "Java Archive" (JAR) file.

In their third role, closely related to the first, packages provide a namespace mechanism to resolve the class name conflicts that may arise when you combine libraries from different providers (Meyer 2009).

Sun Microsystems (1995) gave the points below to list the advantages of Java:

Simple Java omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. These omitted features primarily consist of operator overloading (although the Java language does have method overloading), multiple inheritance, and extensive automatic coercions.

Auto Garbage Collection We added auto garbage collection thereby simplifying the task of Java programming but making the system somewhat more complicated. A good example of a common source of complexity in many C and C++ applications is storage management: the allocation and freeing of memory. By virtue of having automatic garbage collection the Java language not only makes the programming task easier, it also dramatically cuts down on bugs.

Small Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines.

Object-Oriented Simply stated, object-oriented design is a technique that focuses design on the data (=objects) and on the interfaces to it. Object-oriented design is also the mechanism for defining how modules "plug and play."

Distributed Java has an extensive library of routines for coping easily with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the net via URLs with the same ease that programmers are used to when accessing a local file system.

Robust Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error prone.

Pointer Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays. This allows subscript checking to be performed. In addition, it is not possible to turn an arbitrary integer into a pointer by casting.

Secure Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems. The authentication techniques are based on public-key encryption.

Interpreted The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. And since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.

Multithreaded Java has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm... Other benefits of multithreading are better interactive responsiveness and real-time behavior. This is limited, however, by the underlying platform: standalone Java runtime environments have good real-time behavior. Running on top of other systems like Unix, Windows, the Macintosh, or Windows NT limits the real-time responsiveness to that of the underlying system.

Dynamic In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment.

Moreover, Java is a general purpose language which is readily available, widely used and can be seen anywhere.

Wegner (1987) mentioned object oriented programming as: object oriented = objects + classes + inheritance. The Java Swing GUI library, which makes massive use of inheritance, is frequently mentioned as a successful example of software that was designed using object-orientation and it certainly fits Wegner's definition (Ben-Ari, 2010).

The old way of interacting with computers via text-based screen has long gone. Application programs that run on a desktop computer today are mostly GUI based. Teaching programming to students without teaching them GUI does not make sense anymore. However, event-driven programming in GUI-based applications add a level of complexity for the students. The concept of asynchronous events used in a GUI program deviates from the norm of sequential execution of statements in a non-GUI program. A student who is to write a Java GUI program may be overwhelmed by new concepts such as interfaces, listeners, and events (Grissom, 2004), but still will learn everything from the basics and move step by step towards getting the first GUI on the screen, which is the enthusiastic way to teach a student everything from the beginning.

2.3.4. DISADVANTAGES OF JAVA

Dispite the advantages explained above, there are, of course, some disadvantages of Java as the first programming language, too.

Clark (1998), Cecchi (2003) and Cakir (2010) indicated that to write a simple "Hello World" program, a new student may be puzzled with unfamiliar concept like access modifiers, static method, class variable, package, return types, arrays, etc. Hadjerrouit (1998) emphasizes that the syntax of the basic constructs in Java is not easy for novices and that there are complex issues such as file management and multi-threading. He offers that it is more suitable for teaching students with some programming knowledge, particularly in C/C++. However, he agrees that, because of the new possibilities it opens up, it is impossible to ignore the Java paradigm in computer science education. And he adds that the students are enthusiastic about Java, especially for its use in combination with the WWW and game programming for portable devices.

Indeed, working with the Web to execute Java programs adds a certain excitement to the programming process, which further motivates students to learn Java. Obviously, this motivational aspect should not be underestimated, since the use of a language that students enjoy fosters the teaching/learning process and increases the students' acceptance of the language (Cecchi, 2003).

We believe that if the student is exposed to the right question or the right example at the right time, he has nothing to worry about with the syntax or anything else. Examples play an important role in teaching and learning programming. Students as well as teachers cite examples as the most helpful resource for learning to program (Lahtinen, et. al., 2005).

Examples work as role models; students use examples as templates for their own work. Examples must therefore be consistent with the principles and rules being taught and should not exhibit any undesirable properties or behavior. In other words, all examples should follow the very same principles, guidelines, and rules we expect our students to eventually learn. If our examples do not do so consistently, students will have a difficult time recognizing patterns and telling an example's surface properties from those that are structurally or conceptually important. In other words, it is important to present examples in a way that conveys their "message", but at the same time be aware of what learners might actually see in an example (Mason, et. al., 1984).

According to the 40 introductory programming books' authors, they all conclude that examples should concisely illustrate a technique. They should include line numbers for reference, though should preferably be as self-contained as possible, not requiring the reader to keep referring back to the accompanying text discussion. Better examples will often include the author's comments maybe accompanied with some lines and arrows like the typical classroom blackboard example (de Raadt, et. al., 2005).

2.4. ECLIPSE AS THE JAVA EDITOR

Programmers at all levels of experience need to work within environments which give them access to the tools which they must use to accomplish their tasks. This implies that an environment must provide the capability to build and execute a program. For Java, the most basic environment would consist of a simple text editor for editing Java files and a Java Software Development Kit which provides command line tools to compile and execute programs (Pears, et. al., 2007).

A tool that is very adequate for the introduction phase due to its simplicity, could be inappropriate later when more complex concepts are discussed. On the other hand, a tool that is very beneficial in later stages, like for example Eclipse with its many special-purpose plug-ins, may very well interfere with learning in early stages, in case its unnecessary complexity cannot be suppressed (Börstler, et. al., 2008).

According to Börstler and Hadar (2008) an editor should fit the following rules:

- Keep it simple. Tools and examples should be as simple as possible, but still powerful or complex enough to facilitate doing or understanding things that would otherwise have been too difficult for the students.

- Make it sufficiently complex. Examples should be as simple as possible, but not simplistic. Many advantages of the object-orientation paradigm require a certain amount of complexity to become apparent. Example programs need therefore be sufficiently complex to reveal these advantages.

- Make sure it suits your students. There are no “one size fits all” tools and examples; they must be carefully chosen with respect to student background and prerequisite knowledge.

- Make abstract concepts concrete, but don’t stay at the concrete level. Abstract concepts are easier to understand when they are made concrete. However, when staying at a concrete level throughout, students will only get an instrumental understanding of the subject.

- Don’t reinvent the wheel. There are numerous tools and examples “out there” that have been successfully applied in a wide range of settings. However, when reusing a tool or example make sure to evaluate the context of its use

Supporters hoped an IDE would make Java more competitive with Microsoft’s popular Visual Studio .NET, which provides an environment for integrated, easy-to-use software tools that appeal to the many business application developers who aren’t hard-core programmers. This has set off a battle among several Java IDEs, including Borland’s JBuilder, Microsoft’s Visual J#, Oracle’s JDeveloper, and Sun’s NetBeans.

One contender has been Eclipse, which IBM developed and turned over in 2001 to the nonprofit Eclipse Foundation to manage as an open-source platform (Geer, 2005).

Dexter (2007) pointed out the followind ideas about Eclipse IDE which overlaps with the necessary conditions about an editor which Börstler and Hadar mentioned above.

- Eclipse provides a number of aids that make writing Java code much quicker and easier than using a text editor. This means that you can spend more time learning Java, and less time typing and looking up documentation.
- The Eclipse debugger and scrapbook allow you to look inside the execution of the Java code. This allows you to "see" objects and to understand how Java is working behind the scenes
- Eclipse provides full support for agile software development practices such as test-driven development and refactoring. This allows you to learn these practices as you learn Java.
- If you plan to do software development in Java, you'll need to learn Eclipse or some other IDE. So learning Eclipse from the start will save you time and effort.

Object Technology International developed the Java-based technology behind Eclipse before IBM bought the company in 1996. IBM began working on Eclipse internally in 1998 to integrate its many development programs. IBM designed the Eclipse platform in accordance with standards set by the Object Management Group (www.omg.org), which produces and maintains specifications for interoperable enterprise applications. Although the Eclipse Foundation now manages the platform, nonmembers can also build applications using the technology.

Like other IDEs, Eclipse is a programming environment packaged as an application. It consists of a code editor, compiler, debugger, GUI builder, and other tools.

Eclipse is built with Java and thus runs on multiple platforms. However, it will also help build applications in other languages such as C, C++, Cobol, and HTML.

Although it appears to be the Java IDE of choice, Eclipse still faces competition from alternatives such as JBuilder, Visual J#, JDeveloper, and NetBeans (Geer, 2005).

Said Oracle's Farrell, "Eclipse's success is tied to how good a product it is. If it starts to deviate from the main development base, it will begin to lose favor. Now that Eclipse is expanding, there are a lot more people contributing different types of technologies to it. As the base starts to grow, there is a danger of it losing some of its appeal as being lightweight, fast, and focused on the developer."

Murphy, Kersten, and Findlater (2006) made an interesting analysis of the Eclipse users working in the industry. Many software developers spend their workday in an integrated

development environment. They asked 99 developers about the windows they use on the Eclipse Classic (Figure 3).

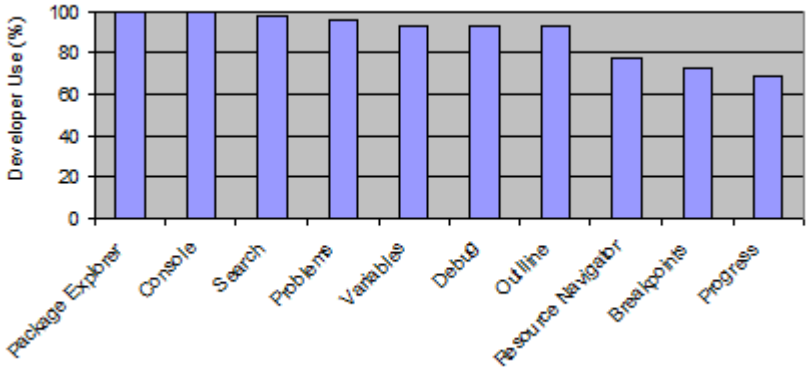

Figure 3 - Eclipse Window Views

And the most popular commands they use are given in Table 5 below.

Table 5 - Most Popular Eclipse Shortcuts

Command	Key Binding	No. of users
Search for references to selected element in workspace	Ctrl+Shift+G	33
Navigate to a type	Ctrl+Shift+T	28
Open a type in the hierarchy view	F4	27
Open declaration of selected element	F3	26
Navigate to last edit location	Ctrl+Q	20
Navigate back among open editors	Alt+Left	19
Search for declarations of selected element in workspace	Ctrl+G	17
Navigate forward among open editors	Alt+Right	14

3. OUTLINE OF THE SE SYLLABUS

"Introduction to programming (SE1)" and "Object Oriented Programming with Java (SE2)" are the courses given under the Software Engineering department during 2 semesters of the first year. The courses consist of a theroretical and a practical session. Each semester, generally, the students have 2 midterms, 1 lab exam, 4 in-class quizzes, and a final exam. 80% attendance on the lab and 70% attendance on the theoretical sessions are obligatory.

3.1. FALL SEMESTER

This semester focuses on algorithms, Java basics, how they work and how they are compiled by the JVM, Eclipse environment and so on. The main topics are as follows:

- Algorithm representations with pseudo code and flow chart
- Algorithm representations using control structures, repetition
- Anatomy of a simple Java program, Java byte codes, Java compiler and Java virtual machine (JVM), Java syntax.
- Basic variables, scope, variable assignment and arithmetic operators, running a Java program both using Eclipse IDE and using the command prompt.
- Logical operators, decision structures, if/else and switch/case blocks
- While, do/while and for loops
- String class and manipulation functions
- Predefined libraries and their functions, function definitions and parameters, function prototype
- Using the Math library and random number generation
- Creating, accessing and using arrays, basic sorting and searching algorithms

3.2. SPRING SEMESTER

This semester fully focuses on object orientation topics, encapsulation, inheritance, abstract structures and interfaces. We first make a review of the first semester by writing complex problems for the first 2 weeks and then start OOP.

- Basics of classes, member variables, class methods, constructors
- Inheritance, polymorphism, class abstraction
- Exception handling
- File I/O
- Java swing components, graphical development
- Basic data structures, list implementations, dynamic allocation

4. INFORMATION OF THE DATA

During 2007 Fall – 2008 Spring season, we had 73 students in the fall semester and 61 in the spring semester. These 73 students will be analyzes considering their family informations, sex, scholarships, parental educations, university gpa's, high school gpa's and high school types, their stay in the university, and their coding skills considering our education. 3 out of this 73 students didn't attend the final exam, hence they will automatically be ignored from the data output.

4.1. ANALYSIS OF THE FIRST EXAM

The students had their first midterm on week 7, which was the week we started showing the String class. We had a overall session to review what we had till the 8th week, which was like a problem session where the students asked questions about their uncertainties and we gave extra questions to help them exercise more.

The first midterm covered the topics; algorithm representations using pseudo code and flowchart, basic Java syntax, basic variables, variable declaration and initialization, scope and curly braces, arithmetic operations, logical operators, decision structures (if-else & switch-case blocks), and loops (while, do-while-for).

The first midterm had 5 questions. The first question was a problem in which the students were asked to get an input from the user, process it according to given conditions, and give a proper output back to the user. 59 out of 70 students found the right solution the the first question, and 6 other got close enough.

As we mentioned above, Caspersen (2006) wrote the importance of writing a proper pseudo code and finding alternative representations of the code. Another question was to convert the given for loop into a while (or do-while) correctly. 52 out of these 59 students made the right conversion to the loops.

There was also a question about converting the if-else statement into a switch-case statement. 60 out of 70 students wrote the right answer for the "if-else" question, and 58 of them made the correct conversion into switch-case, which makes a success rate of 97%, quite a success.

The third question was a bit like the first pseudo code question, with different words but using the same logic. The only difference was that the student was now asked to write the Java code of the given problem. Again, 50 out of these 52 students wrote exactly what the teacher wanted.

The other questions were debugging a given code and giving an output of a given loop segment.

In conclusion, 59 out of 70 (84%) students wrote a proper pseudo code and 50 out of 59 students (85%) gave us what we needed as the pseudo code in the first exam.

The other success to the procedural approach is that it suggested the student should make the right conversion to a given statement, i.e. find correct alternative representation. 52 out of 59 found the correct alternative representation for the loops and 59 out of 60 found the correct alternative for decision structures, which makes 88% and 97% respectively.

4.2. ANALYSIS OF THE SECOND EXAM

Out of 70 students, with one withdrawing the course, 65 took the second midterm. This midterm was largely about functions, their prototypes, and their return values. Topics it covered in addition to the first midterm was the String class, functions and methods, predefined libraries and their predefined functions (e.g. Scanner, Random, Math), creating and accessing arrays, sorting and searching elements on the arrays.

87% of the students who found the right solution to the comparison of 2 arrays, made sorting algorithm problems correctly.

96% of the 65 students wrote the correct function prototype for the given function definition. This was a “must” part in the Caspersen (2006) steps: “Step 6: Method Implementation”.

To implement a method, the student should first find the rigth prototype for the function, i.e. the right return type, necessary input parameters and the naming should be given according to the definition. 96% of success in prototyping was much more than we expected. But function implementation doesn’t end with just writing the correct prototype. This is a huge evidence that the inductive procedural approach is working when the student gets everything gradually.

Continuing the function prototype, the student should implement the function body and test his implementation. According to our second midterm, 89% of the students out of the ones whou found the correct prototype for a given function definition, implemented the correct function body according to the problem.

4.3. ANALYSIS OF THE FINAL EXAM

The final exam was taken by the students after a 14 weeks of an intorductory programming education. The topics covered in the final exam, in addition to the first and the second exam, were complex arrays, multidimensional functions, String operations, and exception handling.

According to what Caspersen (2006), Burton (2003), Grissom (2004), and Cakir (2010) suggested, OOP should go through the definition of the problem (writing a pseudo code), creation of a class, defining its instance fields and empty functions, creation of tests, and method implementation (and maybe creating alternative representations for the methods) steps.

So far, we have seen that writing the pseudo code, defining functions and finding alternative representation, and creating tests over the method steps were successfully accomplished. The only thing left not tested is the “creation ot the class” step.

70 students attended the final examination. Out of these 70 students, 4 students almost made nothing in the exam, hence they won’t be taken into consideration, which gives us a sum of 66 students.

The first question was about String and its predefined functions such as concat(String), indexOf(char), charAt(int), subString(int, int), and compareTo(String). The question was in the form of “fill in the blanks” accoding to the given String. The number of students who filled every gap with the correct answer was 56 out of 66, which makes 85%.

The second question was an exception handling and output question. The student had to catch the given exceptions (ArithmeticException, IndexOutOfBoundsException, Exception) and build a finally block according to the given statement. 55 out of 66 students, 83%, handled the exception correctly, built a finally block, and gave a proper answer.

The third question was a bit complex one, just giving the definition, everything was up to the student. He had to find the correct function prototype, declare and initialize a 2D array, build nested loops on the array, write the correct conditionals, and break the loops. 61 out of 66, 92%, made it all right.

The fourth one was, again, a function definition which asked for the student to shuffle an array of char elements. The question asked the student to use Random class, char array, loops and swapping array elements. 86% found the correct answer to swapping array elements after they correctly built the function prototype, creating the right loop, and using the Random object.

The last question was an OOP question in which the student had to build a Vector class according to the given instructions. There were some class variables, one empty, one full constructors, and 2 functions. Again, we almost had the same results. 98% built the correct class body, 96% built the constructors right, 86% wrote the correct definitions for the functions

5. HYPOTHESES AND RESULTS

The data we had were complex and needed some functional analysis to work on it, that's why we used a statistical tool to analyze the data and get an output from it. The tool we used was SPSS, which is an IBM statistics tool (www.spss.com).

IBM SPSS Statistics is a comprehensive, easy-to-use set of predictive analytic tools for business users, analysts and statistical programmers. For more than 40 years, organizations of all types have relied on IBM SPSS Statistics to increase revenue, outmaneuver competitors, conduct research and make better decisions.

IBM SPSS Statistics offers a broad range of statistical and analytical capabilities that organizations require. It's an easy-to-use, comprehensive software solution that:

- Addresses the entire analytical process from planning and data preparation to analysis, reporting and deployment
- Provides tailored functionality and custom interfaces for different skill levels and functional responsibilities of business users, analysts and statisticians
- Includes flexible deployment options from stand-alone desktop to enterprise-strength server versions
- Provides faster performance and more accurate results, compared to non-statistical, spreadsheet-type software
- Works with all common data types, external programming languages, operating systems and file types
- Offers a broad range of specialized techniques to speed productivity and increase effectiveness

We wanted to see which of the following affected a student's success in learning an object oriented programming language and which affected his success in general, i. e., we evaluated his SE1 and SE2 scores in addition to his general GPA.

The attributes were as follows: gender, major, scholarship status, high school type, education duration (the time he spent in the univerity), mother & father education degree and their working status, duration of waiting (the time he waited before getting into a university), high school GPA (out of 5), and where he stays.

Hypotheses that we tried to evaluate were as follows;

H1: Gender has an effect on the students' capability of Java learning.

H2: Scholarship has affected the students' achivements on Java learning.

H3: ÖSS (University Entrance Exam for Students) degree has an effect on the students' success.

H4: Parents' educational degree and working status has affected the achievements.

H5: Students' place of residence has affected their learning.

H6: Their duration of study in the university affected their success in learning Java.

H7: Their success increases as the time they spend before getting into a university increases.

H8: High school type and high school GPA effect the students' success.

The method used to analyze the data was a bivariate correlation, because there were multiple attributes that we tried to find the effects on the data. Pearson and Spearman correlation coefficients were used with two-tailed test of significance.

The descriptive statistics (mean and standard deviation of the attributes) of the output can be seen in Table 6.

Table 6 - Descriptive Statistics

	Mean	Std. Deviation	N
gender	,39	,492	76
major	,62	,692	76
scholarship	31,58	43,850	76
general GPA	2,5090	,62310	73
JAVA	2,4346	,86067	76
duration of education	4,34	,888	76
ÖSS degree	1,86	1,363	76
father's education	2,17	1,182	76
mother's education	1,64	1,283	76
father's working status	3,48	2,220	75
mother's working status	1,22	2,044	76
time waited before UNI	,93	1,112	76
high school type	1,16	1,255	76
high school GPA	2,82	,875	76
residence	7,70	3,812	76

Table 7 shows Java and cumulative GPA values of the attributes according to the hypotheses.

Table 7 – Pearson Correlation Coefficients about Java achievements and GPA

Attribute	Java Learning	Cumulative GPA
Gender	.156	.314
Scholarship	.468	.471
ÖSS	-.556	-.540
Mother's Education	.185	.113
Father's Education	.214	.156
Mother's Working Status	.053	.028
Father's Working Status	.188	.281
Residence	.065	.114
Duration of Education	-.366	-.405
Time Spent before UNI	-.142	-.016
High School Type	.030	.019
High School GPA	.202	.277

It can be seen from Table 7 that gender has no any effect which is almost none on Java learning. This eliminates hypothesis number 1: "Gender has an effect on the students' capability of Java learning." But still, in general GPA scores, females seem to be more successful than males.

Based on the cumulative GPA the females are more successful than males (2.75 / 2.25 out of 4.00) and their average duration of education is less than males (4.2 years / 4.54 years).

The second attribute, major, has no effect on the students' enthusiasm on Java learning, either. There were 4 different majors in out data, Computer Engineers, Software Engineers, Math&Computer Scientists, and Computer&Instructional Technologists. The constant was 0.148, which cannot be taken into consideration.

One of the expected results came to be true with the scholarship attribute. In Bahçeşehir University, scholarship of the student doesn't change, i.e. the student never loses his scholarship. He has the chance to work hard, get a scholarship from the university and then stop working as soon as he enters the university. But our results show that it wasn't that way, on the contrary, the student's scores rise as his scholarship status rises. Scholarship attribute has an effect on both Java learning and general GPA, which have the constants 0.471 and 0.468 respectively. It can be concluded that hypothesis number 2; "Scholarship has affected the students' achivements on Java learning." Came out to be true.

The second expected result came from the general GPA. As the general GPA increases Java learning increases, and vice versa. The constant was 0.615. But this is common sense, that's why it was not on the hypothesis list.

As the ÖSS degree increases, (the rank of the student among all of the ÖSS students), the students' general GPA and Java scores decrease. This result is closer to the scholarship result. If the degree decreases (i. e., his rank among others increases), his scholarship and his scores increase. It can be taken into consideration that the ÖSS score has a positive effect on the students' duration of education (with a correlation of 0.394). As the student's rank becomes lower, his duration of education gets longer and longer. These prove that the 3rd hypothesis is correct: "ÖSS (University Entrance Exam for Students) degree has an effect on the students' success."

Students' parents' working status and educational degree and where they stay (residence) seem to have nothing to do with their general GPA and Java scores, so this eliminates the 4th hypothesis: "Parents' educational degree and working status has affected the achievements." and our 5th hypothesis: "Students' place of residence has affected their learning."

Duration of education has a inverse effect on both Java learning and general GPA, with the constants -0.405 and -0.366 respectively. As the time the student spends in the university increases, his scores decrease, which was suprising because the student has more time to enroll in the same course and increase his score. It seems that our students didn't choose to increase their programming language scores when they had other courses. This proves our 6th hypothesis: "Their duration of study in the university affected their success in learning Java."

High school GPA, high school type, and type waited before getting in the university seems to be effectless on both Java learning and general GPA of the students. These eliminate the 7th and 8th hypotheses, "Their success increases as the time they spend before getting into a university increases" and "High school degree, high school GPA,and high school type has an effect on the students' success."

To sum up, Table 8 gives an outline of the hypotheses and their correctness:

Table 8 - Hypotheses

No.	Hypothesis	Coefficient	Correctness
H1	Gender has an effect on the students' capability of Java learning.	0.156	✗
H2	Scholarship has affected the students' achivements on Java learning.	0.468 ($p<0.01$)	✓
H3	ÖSS (University Entrance Exam for Students) degree has an effect on the students' success.	-0.556 ($p<0.01$)	✓
H4	Parents' educational degree and working status has affected the achievements.	0.185 / 0.214	✗
H5	Students' place of residence has affected their learning.	0.065	✗
H6	Their duration of study in the university affected their success in learning Java.	-0.366 ($p<0.05$)	✓
H7	Their success increases as the time they spend before getting into a university increases.	-0.142	✗
H8	High school type and high school GPA effect the students' success.	0.030 / 0.202	✗

The correlation table can be seen on Table 9:

Table 9 - Correlation

		gender	major	scholarship	general GPA	JAVA	duration of education	OSS degree	father's education	mother's education	father's working status	mother's working status	time waited before UNI	high school type	high school GPA	residence
gender	Pearson Correlation	1	,331**	-,091	,314**	,156	-,130	,047	,180	,141	-,074	-,062	,000	,157	,109	,086
	Sig. (2-tailed)		,004	,434	,007	,178	,263	,690	,119	,226	,531	,592	,996	,176	,348	,461
	Sum of Squares and Cross-products	18,158	8,447	-147,368	6,848	4,962	-4,263	2,342	7,868	6,658	-5,920	-4,711	-,026	7,263	3,526	12,079
	Covariance	,242	,113	-1,965	,096	,066	-,057	,031	,105	,089	-,080	-,063	,000	,097	,047	,161
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
major	Pearson Correlation	,331**	1	,227*	,370**	,148	,194	-,130	,000	,041	,139	,089	-,224	,009	,323**	,067
	Sig. (2-tailed)	,004		,049	,001	,203	,094	,263	,996	,728	,233	,442	,052	,939	,004	,566
	Sum of Squares and Cross-products	8,447	35,934	515,789	11,613	6,594	8,921	-9,197	-,039	2,697	15,920	9,487	-12,908	,579	14,658	13,224
	Covariance	,113	,479	6,877	,161	,088	,119	-,123	,000	,036	,215	,126	-,172	,008	,195	,176
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
scholarship	Pearson Correlation	-,091	,227*	1	,471**	,468**	-,196	-,870**	-,093	-,177	,318**	-,005	-,189	-,104	,432**	-,133
	Sig. (2-tailed)	,434	,049		,000	,000	,090	,000	,426	,126	,005	,963	,101	,372	,000	,250
	Sum of Squares and Cross-products	-147,368	515,789	144210,526	926,158	1,325E3	-571,053	-3902,632	-360,526	-747,368	2298,000	-36,842	-692,105	-428,947	1242,105	-1673,684
	Covariance	-1,965	6,877	1922,807	12,863	17,673	-7,614	-52,035	-4,807	-9,965	31,054	-,491	-9,228	-5,719	16,561	-22,316
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
general GPA	Pearson Correlation	,314**	,370**	,471**	1	,615**	-,405**	-,540**	-,156	-,113	,281*	,028	-,016	,019	,277*	,114
	Sig. (2-tailed)	,007	,001	,000		,000	,000	,000	,186	,341	,017	,816	,891	,873	,018	,337
	Sum of Squares and Cross-products	6,848	11,613	926,158	27,954	22,990	-15,816	-32,660	8,382	6,506	27,579	2,545	-,822	1,037	10,838	19,742
	Covariance	,096	,161	12,863	,388	,319	-,220	-,454	,116	,090	,388	,035	-,011	,014	,151	,274
	N	73	73	73	73	73	73	73	73	73	72	73	73	73	73	73
JAVA	Pearson Correlation	,156	,148	,468**	,615**	1	-,366**	-,556**	-,214	-,185	,188	-,053	-,142	,030	,202	,095
	Sig. (2-tailed)	,178	,203	,000	,000		,001	,000	,064	,109	,105	,649	,222	,795	,080	,578
	Sum of Squares and Cross-products	4,962	6,594	1325,447	22,990	55,556	-20,950	-48,909	16,300	15,354	26,817	-7,008	-10,177	2,455	11,404	15,956
	Covariance	,066	,088	17,673	,319	,741	-,279	-,652	,217	,205	,362	-,093	-,136	,033	,152	,213
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
duration of education	Pearson Correlation	-,130	,194	-,196	-,405**	-,366**	1	,394**	,020	-,009	-,041	,141	-,207	,118	-,107	-,123
	Sig. (2-tailed)	,263	,094	,090	,000	,001		,000	,866	,939	,727	,224	,073	,308	,359	,291
	Sum of Squares and Cross-products	-4,263	8,921	-571,053	-15,816	-20,950	59,105	35,763	1,553	-,763	-6,000	19,184	-15,289	9,895	-6,211	-31,132
	Covariance	-,057	,119	-7,614	-,220	-,279	,788	,477	,021	-,010	-,081	,256	-,204	,132	-,083	-,415
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
OSS degree	Pearson Correlation	,047	-,130	-,870**	-,540**	-,556**	,394**	1	,115	,168	-,340**	,069	,187	,060	-,356**	,091
	Sig. (2-tailed)	,690	,263	,000	,000	,000	,000		,323	,146	,003	,552	,105	,605	,001	,485
	Sum of Squares and Cross-products	2,342	-9,197	-3902,632	-32,660	-48,909	35,763	139,408	13,882	22,092	-76,240	14,461	21,276	7,737	-32,026	31,671
	Covariance	,031	-,123	-52,035	-,454	-,652	,477	1,859	,185	,295	-1,030	,193	,284	,103	-,427	,422
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
father's education	Pearson Correlation	,180	,000	-,093	,156	,214	,020	,115	1	,603**	,047	,157	,177	,170	,018	,089
	Sig. (2-tailed)	,119	,996	,426	,186	,064	,866	,323		,000	,689	,178	,126	,141	,877	,447
	Sum of Squares and Cross-products	7,868	-,039	-360,526	8,382	16,300	1,553	13,882	104,776	68,618	30,760	32,092	-14,145	18,947	1,395	29,934
	Covariance	,105	,000	-4,807	,116	,217	,021	,185	1,397	,915	,416	,428	-,189	,253	,019	,399
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
mother's education	Pearson Correlation	,141	,041	-,177	,113	,185	-,009	,168	,603**	1	,047	,519**	-,213	,118	,107	,013
	Sig. (2-tailed)	,226	,728	,126	,341	,109	,939	,146	,000		,689	,000	,065	,309	,357	,910
	Sum of Squares and Cross-products	6,658	2,697	-747,368	6,506	15,354	-,763	22,092	68,618	123,408	9,980	102,039	-22,776	14,263	9,026	4,829
	Covariance	,089	,036	-9,965	,090	,205	-,010	,295	,915	1,645	,135	1,361	-,304	,190	,120	,064
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
father's working status	Pearson Correlation	-,074	,139	,318**	,281*	,188	-,041	-,340**	,157	,047	1	,166	,029	-,171	,314**	-,040
	Sig. (2-tailed)	,531	,233	,005	,017	,105	,727	,003	,178	,689		,155	,802	,143	,006	,735
	Sum of Squares and Cross-products	-5,920	15,920	2298,000	27,579	26,817	-6,000	-76,240	30,760	9,980	364,720	54,760	5,400	-35,240	45,240	-25,000
	Covariance	-,080	,215	31,054	,388	,362	-,081	-1,030	,416	,135	4,929	,740	,073	-,476	,611	-,338
	N	75	75	75	72	75	75	75	75	75	75	75	75	75	75	75
mother's working status	Pearson Correlation	-,062	,089	-,005	,028	-,053	,141	,069	,177	,519**	,166	1	-,187	-,009	-,021	-,121
	Sig. (2-tailed)	,592	,442	,963	,816	,649	,224	,552	,126	,000	,155		,106	,940	,854	,297
	Sum of Squares and Cross-products	-4,711	9,487	-36,842	2,545	-7,008	19,184	14,461	32,092	102,039	54,760	313,197	-31,882	-1,684	-2,868	-70,855
	Covariance	-,063	,126	-,491	,035	-,093	,256	,193	,428	1,361	,740	4,176	-,425	-,022	-,038	-,945
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
time waited before UNI	Pearson Correlation	,000	-,224	-,189	-,016	-,142	-,207	,187	-,144	-,213	,029	-,187	1	-,317**	-,067	-,074
	Sig. (2-tailed)	,996	,052	,101	,891	,222	,073	,105	,216	,065	,802	,106		,005	,563	,525
	Sum of Squares and Cross-products	-,026	-12,908	-692,105	-,822	-10,177	-15,289	21,276	-14,145	-22,776	5,400	-31,882	92,671	-33,211	-4,921	-23,513
	Covariance	,000	-,172	-9,228	-,011	-,136	-,204	,284	-,189	-,304	,073	-,425	1,236	-,443	-,066	-,314
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
high school type	Pearson Correlation	,157	,009	-,104	,019	,030	,118	,060	,170	,118	-,171	-,009	-,317**	1	-,119	-,032
	Sig. (2-tailed)	,176	,939	,372	,873	,795	,308	,605	,141	,309	,143	,940	,005		,306	,786
	Sum of Squares and Cross-products	7,263	,579	-428,947	1,037	2,455	9,895	7,737	18,947	14,263	-35,240	-1,684	-33,211	118,105	-9,789	-11,368
	Covariance	,097	,008	-5,719	,014	,033	,132	,103	,253	,190	-,476	-,022	-,443	1,575	-,131	-,152
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
high school GPA	Pearson Correlation	,109	,323**	,432**	,277*	,202	-,107	-,356**	,018	,107	,314**	-,021	,067	-,119	1	-,193
	Sig. (2-tailed)	,348	,004	,000	,018	,080	,359	,001	,877	,357	,006	,854	,563	,306		,095
	Sum of Squares and Cross-products	3,526	14,658	1242,105	10,838	11,404	-6,211	-32,026	1,395	9,026	45,240	-2,868	-4,921	-9,789	57,421	-48,237
	Covariance	,047	,195	16,561	,151	,152	-,083	-,427	,019	,120	,611	-,038	-,066	-,131	,766	-,643
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76
residence	Pearson Correlation	,086	,067	-,133	,114	,065	-,123	,091	,089	,013	-,040	-,121	-,074	-,032	-,193	1
	Sig. (2-tailed)	,461	,566	,250	,337	,578	,291	,485	,447	,910	,735	,297	,525	,786	,095	
	Sum of Squares and Cross-products	12,079	13,224	-1673,684	19,742	15,956	-31,132	31,671	29,934	4,829	-25,000	-70,855	-23,513	-11,368	-48,237	1090,039
	Covariance	,161	,176	-22,316	,274	,213	-,415	,422	,399	,064	-,338	-,945	-,314	-,152	-,643	14,534
	N	76	76	76	73	76	76	76	76	76	75	76	76	76	76	76

6. CONCLUSION

The aim of this study was to analyze the effect of the learning methodologies in order we used on the students by examining the objects learned by the students.

Our teaching metholodogy was the inductive procedural approach to object oriented programming and according to our detailed exam evaluations, this method was successful with an approximately 88% ratio.

In addition to the exam assesments, the students' personal information was taken into account which covered gender, major, scholarship status, high school type, education duration (the time they spent in the univerity), mother and father education degree and their working status, duration of waiting (the time they waited before getting into a university), high school GPA (out of 5), and which city they come from.

A bivariate correlation analysis was executed using the SPSS tool on the data and other than the expected results, some suprising outcomes raised.

The scholarship had a positive and the duration of education and ÖSS degree had a negative effect on the success of the students as we concluded. Gender had no effect on Java learning but females were more successfull than men in general GPA.

All in all, it was concluded that the inductive procedural approach on teaching object oriented programming is successful among other personal factors such as the student's scholarship status and duration of education.

Further work may be done by examining a study group in which the same education is given on an objects-first manner rather than a procedural approach, compare the results with this work's results, and decide which approach is better for the student.

REFERENCES

Publications

Arif, E. M., 2000, *A Methodology for Teaching Object-Oriented Programming Concepts in an Advanced Programming Course*, SIGCSE Bulletin, **Vol. 32, No. 2**

Bellamy, R. K. E., 1994, *What Does Pseudo-Code Do? A Psychological Analysis of the Use of Pseudo-Code by Experienced Programmers*, Lawrence Eribaum Associates, Inc., HUMAN-COMPUTER INTERACTION, **Vol. 9, pp. 225-246**

Ben-Ari, M., 2010, *Objects Never? Well, Hardly Ever!*, Communications of the ACM, **Vol. 53, No. 9**

Briot, J.-P., Guerraoui, R., Lohr, K. P., 1998, *Concurrency and Distribution in Object-Oriented Programming*, ACM Computing Surveys, **Vol. 30, No. 3**

Börstler, J., Hadar, I., 2007, *Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts*, Springer Verlag, ECOOP 2007 Workshop Reader, LNCS 4906, **pp. 182–192**

Börstler, J., Hall, M. S., Nordström, M., Paterson, J. H., Sanders, K., Schulte, C., Thomas, L., 2009*, An Evaluation of Object Oriented Example Programs in Introductory Programming Textbooks*, SIGCS Bulletin, **Vol. 41, No. 4**

Burton, P. J., Bruhn, R. E., 2003, *Teaching Programming in the OOP Era*, SIGCSE Bulletin, **Vol. 35, No. 2**

Cakir, D., Kaptan, S. N., Karahoca, A., 2010, *An OOP w/ Java Course Using an Inductive Approach*, WCE 2010, June 30 - July 2, 2010, London, U.K., Proceedings of the World Congress on Engineering 2010, **Vol. 1**

Caspersen, M. E., Kölling, M., 2006, *A Novice's Process of Object-Oriented Programming*, OOPSLA '06, October 22 – 26, Portland, Oregon, USA

Cecchi, L., Crescenzi, P., Innocenti, G., 2003*, C : C++ = JavaMM : Java (A Simple Tool for Teaching Java in a CS1 Course)*, PPPJ 2003, 16 – 18 June 2003, Kilkenny City, Ireland

Clark, D., MacNish, C., 1998, *Java as a Teaching Language – Opportunities, Pitfalls and Solutions*, Proceedings of the third Australasian Conference on Computer Science Education, **pp.173-179**

Cross, J. H., Sheppard, S. V., 1988, *Graphical Extensions for Pseudo-Code, PDLs, and Source Code*, Proceedings of the ACM 16th Annual Conference on Computer Science (Atlanta, GA), New York, **pp. 520-528**

de Raadt, M., Watson, R., Toleman, M., 2005, *Textbooks: Under Inspection*, Technical Report, University of Southern Queensland, Department of Maths and Computing, Toowoomba, Australia

Deimel, L. E., Neveda, J. F., 1990, *Reading Computer Programs: Instructor's guide and exercises*, CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

Dexter, M., 2007, *Eclipse and Java for Total Beginners – Companion Tutorial Document*, Licensed under the Educational Community License **v1.0**

Franca, P. B., 1994, *Software Engineering Education – The Shift to Object Oriented Programming*, IEEE

Geer, D., 2005, *Eclipse Becomes the Dominant Java IDE*, IEEE Computer Society

Goldwasser, M. H., Letscher, D., 2008, *Teaching an Object-Oriented CS1 – With Python*, ITiCSE'08, June 30–July 2, Madrid, Spain

Grissom, S., Dulimarta, H., 2004, *An Approach to Teaching Object Oriented Design in CS2*, Consortium for Computing Sciences in Colleges, JCSC **20, 1**

Hadjerrouit, S., 1998, *Java as First Programming Language: A Critical Evaluation*, SIGCSE Bulletin, **Vol. 30, No. 2**

Hamilton, M., Haywood, L., 2004, *Learning About Software Development – Should Programming Always Come First?*, 6th Australasian Computing Education Conference (ACE2004) Dunedin, New Zealand, **Vol. 30**

Hu, C., 2004, *Rethinking of teaching object-First*, Education and Information Technologies, **Vol. 9, No. 3, pp. 209-218**

King, K. N., 1997, *The Case for Java as a First Language*, Proceedings of the 35th Annual ACM Southeast Conference (April 1997), **pp. 124–131**

Kölling, M., 1999, *The Problem of Teaching Object-Oriented Programming, Part 1: Languages*, Journal of Object-Oriented Programming, **Vol. 11, No. 8, pp. 8-15**

Kölling, M., Koch, B., Rosenberg, J., 1995, *Requirements for a First Year Object-Oriented Teaching Language*, SIGCSE '95 3/95, Nashville, TN, USA

Kristensen, B. B., Osterbeye, K., 1996, A *conceptual perspective on the comparison of object-oriented programming languages*, ACM SIGPLAN, **Vol. 31, No. 2**

Lahtinen, E., Ala-Mutka, K., Jarvinen, H., 2005, *A Study of the Difficulties of Novice Programmers*, Proceedings of the 10th Annual SIGCSE Conference on Innovation and

Technology in Computer Science Education, **pp. 14–18**

Leavens, G. T., 1990, *Introduction to the Literature on Object-Oriented Design, Programming, and Languages*, CACM, **Vol. 33, No. 9**

Lewis, J., 2000, *Myths About Object-Orientation and Its Pedagogy*, SIGCSE 2000 3/00, Austin, TX, USA

Li, D. X., Prasad, C., 2005, *Effectively Teaching Coding Standards in Programming*, SIGITE '05, October 20 – 22, Newark, New Jersey, USA

Mannila, L., de Raadt, M., 2006, *An Objective Comparison of Languages for Teaching Introductory Programming*, Proceedings Koli Calling

Mason, J., Pimm, D., 1984, *Generic Examples: Seeing the General in the Particular*, Educational Studies in Mathematics, **Vol. 15, No. 3, pp. 227-289**

McKim, Jr., J. C., Ellis, H. J. C., 2004, *Using a Multiple Term Project to Teach Object Oriented Programming and Design*, CSEET '04, IEEE

Meyer, B., 2009, *An Introduction to Java (from Material by Marco Piccioni)*, SpringerLink, Touch of Class, **Part 6, pp. 747-774**

Müller, B., 1993*, Is Object-Oriented Programming Structured Programming?,* ACM SIGPLAN Notices, **Vol. 28, No . 9**

Murphy, G. C., Kersten, M., Findlater, L., 2006, *How Are Java Software Developers Using the Eclipse IDE?*, IEEE Computer Society

Nassi, I., Shneiderman, B., 1973, Flowchart Techniques for Structured Programming, SIGPLAN **notices 8, 8, 12-26**

Northrop, L. M., 1992, *Finding an Educational Perspective for Object-Oriented Development*, OOPSLA '92, 5 – 10 October, Vancouver, British Columbia, Canada

Olsen, A. L., 2005, *Using Pseudocode to Teach Problem Solving*, Consortium for Computing Sciences in Colleges, JCSC **21, 2**

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., Paterson, J., 2007, *A Survey of Literature on the Teaching of Introductory Programming*, ITiCSE, June 2007, Dundee, Scotland

Pedroni, M., Meyer, B., 2006, *The Inverted Curriculum in Practice*, SIGCSE'06, March 1 – 5, Houston, Texas, USA

Pokkunuri, B. P., 1992*, Object Oriented Programming*, SIGPLAN Notices, **Vol. 24, No. 2**

Ragonis, N., 2010, *A Pedagogical Approach to Discussing Fundamental Object-Oriented Programming Principles Using the ADT SET*, ACM inroads, **Vol. 1, No. 2**

Roy, G. G., 2006, *Designing and Explaining Programs with a Literate Pseudocode*, ACM Journal of Educational Resources in Computing, **Vol. 6, No. 1**

Ryder, B. G., 2003, *Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages*, Springer-Verlag, CC 2003, LNCS 2622, **pp. 126–137**

Scanlin, D., 1988, *Should Short, Relatively Complex Algorithms be Taught Using Both Graphical and Verbal Methods*, Proceedings of the ACM SIGCSE, New York, **pp. 185-189**

Schneider, G. M., 1978, *The Introductory Programming Course in Computer Science: Ten Principles*, 9th SIGCSE/CSA Technical Symposium on Computer Science Education, **pp. 107–114**

Snyder, A., 1986, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, OOPSLA '86 Proceedings

Sun Microsystems, 1995, *The Java Language: An Overview*, Copyright © 1994, 1995 Sun Microsystems

Thimbleby, H., 2003, *Explaining Code for Publication*, Software Practice and Experience, **Vol. 33, No. 10, pp. 975-1001**

Wegner, P., 1987, *Dimensions of Object-Based Language Design*, Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications

Zhu, H., Zhou, M., 2003, *Methodology First and Language Second: A Way to Teach Object-Oriented Programming*, OOPSLA'03, October 26–30, Anaheim, California, USA

Books

Baldwin, D. and Scragg, G. W., June 2004, *Algorithms and Data Structures: The Science of Computing*, Charles River Media, Massachusetts, 1584502509

Eckel, B., 1998, *Thinking in Java*, Prentice Hall, New Jersey, 0-13-659723-8

Garrido, J. M., 2003, *Object-Oriented Programming: From Problem Solving to Java*, Charles River Media, Massachusetts, 1-58450-287-8

Martin, J., McLure, C., 1985, *Diagramming Techniques for Analysts and Programmers*, Prentice Hall, Englewood Cliffs, New Jersey, 0-13-208794-4

Orr, K. T., 1977, *Structured Systems Development*, Yourden Press, New York, 0138551499

Savitch, W. December 2003. *Absolute Java*, Addison Wesley, 978-0321205674

Valdo, J., May 2010. *Java: The Good Parts*, O'Reilly, California, 978-0-596-80373-5

Electronic Sources

Eclipse IDE (Classic) – Java Editor, Eclipse Classic 3.6.0, http://www.eclipse.org/ (last visited 20.09.2010)

Tiobe Software – The Coding Standards Company, TIOBE Programming Community Index for September 2010, http://www.tiobe.com/ (last visited 20.09.2010)

RoseIndia, http://www.roseindia.net/ (last visited 20.09.2010)

Statistical Package for Social Sciences, SPSS, IBM SPSS Statistics, www.spss.com (last visited 20.09.2010)

CURRICULUM VITAE

FULL NAME	**:**	Duygu ÇAKIR
ADDRESS	**:**	Enverpaşa Cad. Yakup Cemil Sok. D:39/A D:4 Esenkent-Esenyurt / İstanbul / Türkiye
EMAIL	**:**	duygu.cakir@bahcesehir.edu.tr
BIRTH PLACE / YEAR	**:**	Üsküdar/İstanbul - 1985
LANGUAGE	**:**	Turkish (native), English
HIGH SCHOOL	**:**	Bahçeşehir College
UNIVERSITY	**:**	Computer Engineering**,** Bahçeşehir University, 2007
MSc	:	Bahçeşehir University, 2010
NAME OF INSTITUTE	**:**	Institute of Science
NAME OF PROGRAM	**:**	Computer Engineering
WORK EXPERIENCE	**:**	Bahçeşehir University, Teaching Assistant 2007-2010