Spring 3-23-2018

Assessing Students' Object-Oriented Programming Skills with Java: The "Department-Employee" Project

Tyler T. Redman
University of North Alabama, tredman@una.edu

Mark G. Terwilliger
University of North Alabama, mterwilliger@una.edu

John D. Crabtree
University of North Alabama, jcrabtree@una.edu

Xihui Zhang
University of North Alabama, xzhang6@una.edu

ASSESSING STUDENTS' OBJECT-ORIENTED PROGRAMMING SKILLS WITH JAVA: THE "DEPARTMENT-EMPLOYEE" PROJECT

Tyler T. Redman
University of North Alabama
tredman@una.edu

Mark G. Terwilliger
University of North Alabama
mterwilliger@una.edu

John D. Crabtree
University of North Alabama
jcrabtree@una.edu

Xihui Zhang
University of North Alabama
xzhang6@una.edu

ABSTRACT

Java is arguably today's most popular and widely used object-oriented programming language. Learning Java is a daunting task for students, and teaching it is a challenging undertaking for instructors. To assess students' object-oriented programming skills with Java, we developed the "Department-Employee" project. In this article, we review the history of object-oriented programming and provide an overview of object-oriented programming with Java. We also provide the project specification as well as the course background, grading rubric, and score reports. Survey data are presented on students' backgrounds, as well as students' perceptions regarding the project. Results from the instructor score reports and student perceptions show that the "Department-Employee" project was effective in assessing students' object-oriented programming skills with Java.

Keywords

Structured programming, object-oriented programming, Java

INTRODUCTION

Java is an object-oriented programming language. Compared to structured programming techniques, object-oriented design and programming provides a more natural and intuitive way to describe real-world objects by creating classes and their runtime objects (also called instances). People possessing strong object-oriented programming skills with Java are in high demand in industry. Nine out of today's top ten most popular programming languages support object-oriented programming ("TIOBE index," 2018). Although there are over 100 programming languages in existence today, research has shown that language adoption follows a power law, and the top six languages account for 75% of the software projects at SourceForge (Meyerovich & Rabkin, 2013). As such, it is imperative that institutions of higher learning teach their students object-oriented programming skills so that the students are well prepared for their prospective careers in the field of information technology.

Learning an object-oriented programming language such as Java, especially for novices, can be a daunting task (Cavaiani, 2006). This is because they need to learn not only structured programming concepts and skills, which are used for the code written within Java methods, but also object-oriented concepts and skills, which are used for higher-level abstractions. Furthermore, they also need to learn "how to use the Java Class Library to locate the details of classes, methods, and toolkits that they can use in their own classes" (Cavaiani, 2006, p. 365). Java is said to have a "fairly complicated syntax and fairly complicated static semantics" (Shein, 2015, p. 19). Ramesh and Wu (2004) have identified three main hurdles that students face: (a) inability to clearly distinguish between the notion of classes and objects, (b) understanding and using complex objects (objects that contain one or more objects within them), and (c) grasping the concepts of inheritance and interfaces.

Teaching an object-oriented programming language such as Java can be a challenging undertaking. A key question facing programming instructors is whether to take an objects-first approach or a structures-first approach to the object-oriented programming education. Primitive types in Java (i.e., boolean, char, byte, short, int, long, float, and double) have long been considered a weakness of Java because they violate the principle of orthogonality in programming language design (Ourosoff, 2002). The classic "hello, world" program was also considered harmful because object-oriented programming should be learned naturally, not procedurally (Westfall, 2001). The research results by Johnson and Moses (2008) indicate that students who take an objects-first approach to object-oriented programming outperform those who take a structures-first approach. Hu (2008) maintains that the debate between "objects early" and "objects later" was not that important, and just saying "a class defines a data type" would be adequate. Xing and Belkhouche (2003) argue that critical object-oriented design

should be given precedence over object-oriented programming techniques in teaching object-oriented programming. Furthermore, the use of a modern Integrated Development Environment (IDE) 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 (Pendergast, 2006).

In short, learning object-oriented Java programming is a formidable task for students, and teaching it is a challenging undertaking for instructors. To improve the effectiveness of both instructor teaching and student learning, it is imperative to develop tools to assess students' object-oriented programming skills with Java. In this research, we create a "Department-Employee" project to assist in this process. This project may be used in any Java course in which the proper concepts, as detailed below, have been covered. We are not advocating a particular course design or approach (e.g., objects-first) and, in fact, the authors of this article have different preferences. However, assessment of the students' grasp of object-oriented concepts is important in any Java course, regardless of the particular course design, and this assignment has been a useful assessment tool in our courses.

This article describes the "Department-Employee" project. It begins with a review of the history of object-oriented programming and an overview of object-oriented programming with Java. The specification of the project, the course background, the grading rubric and score reports, as well as the students' perceptions about the project and the students' backgrounds, are also presented.

A BRIEF HISTORY OF OBJECT-ORIENTED PROGRAMMING

Smalltalk, the first true object-oriented programming (OOP) language, has been around since the 1960's. However, it was not until the 1980's that hardware advances, especially in the area of random access memory, made OOP practical enough to gain mainstream acceptance in industry.

Object-oriented programming has always been difficult to define, but the concept of the "object" is certainly central to any definition (Pokkunuri, 1989; Rentsch, 1982). The focus on objects naturally led to object-oriented approaches to analysis and design. Many different approaches to problem decomposition in terms of objects and classes led to the "method wars" of the 1980's and 1990's. Many thought-leaders produced methodologies, along with their associated object notations and/or tools. Three authors who gained preeminence in the early 1990's published the following definitions:

- *Grady Booch (Rational Rose)*: Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships (Booch, 1991, p. 36).
- *James Rumbaugh (Object Modeling Technique)*: Superficially the term "object-oriented" means that we organize software as a collection of discrete objects that incorporate both data structure and behavior. This is in contrast to conventional programming in which data structure and behavior are only loosely connected. There is some dispute about exactly what characteristics are required by an object-oriented approach, but they generally include four aspects: identity, classification, polymorphism, and inheritance (Rumbaugh et al., 1991, p. 1).
- *Ivar Jacobson (Objectory)*: In his chapter titled "What is object-orientation?" Ivar Jacobson states that his aim is "to introduce the actual idea [of object-oriented technology], not to give strict and precise definitions" (Jacobson, 1993, p. 42). It is worth noting that in the 1990's Jacobson joined Booch and Rumbaugh in the collaboration that gave us the Unified Modeling Language (UML) and the Unified Process.

While properly defining objects and their precise characteristics may be difficult, the standard premise of OOP has always been that it should be simpler to work with objects since "people already think in terms of objects" (Weisfeld, 2008, p. 6). However, as Bruce Eckel points out, there are hurdles that the typical programmer must deal with: "If there's a downside, it is the expense of the learning curve. Thinking in objects is a dramatic departure from thinking procedurally, and the process of *designing* objects is much more challenging than procedural design, especially if you're trying to create reusable objects" (Eckel, 1998, p. 25).

In the 1980's the new C++ language, created by Bjarne Stroustrup, gained traction and remains one of the most popular object-oriented programming languages today ("TIOBE index," 2018). This new language, which leveraged the existing robust support for the very popular C language, was originally named C with Classes.

While C++ was certainly popular in the early 1990's (and still is), it has its problems (Meyerovich & Rabkin, 2013; "TIOBE index," 2018). The language was not formally standardized until 1998 as ISO/IEC 14882:1998. While the release of *The Annotated C++ Reference Manual* in 1990 served as a de-facto language standard, all compilers do not follow the same language rules and consequently porting code from one compiler to another is often difficult. In addition, C++ code compiled

for a particular hardware architecture and/or runtime environment cannot be used in a different environment without recompilation from the source code (which may involve using a different compiler).

These and other issues inspired James Gosling at Sun Microsystems to create a language, originally known as Oak, that was object-oriented, standard, and portable. The first releases of the rebranded Java programming language and its Java Development Kit (JDK) were made available to the public in 1995. According to Benander et al. (2003), the best feature of Java is its "platform independency."

OBJECT-ORIENTED PROGRAMMING WITH JAVA

Since it did not need to rely upon an existing language for support (e.g., C), Java was designed from the ground up to be object-oriented. All Java programs are object-oriented. On the contrary, with a language like C++, it is possible for programmers to fall back into a more procedural, C-like programming style. While not all Java programs are guaranteed to have *good* object-oriented design, all Java code must be written in the context of a class and, more specifically, within a method belonging to a class. From a software engineering perspective, object-oriented design is often used in the architectural design, while structured design and programming is used for detailed method implementation (Zhang, 2010). Virtually all of the executable code in a Java class is written within methods, and that code must be structured code.

However, not all variables in a Java program are objects. Performance concerns led the creator of Java to provide "primitive" data types that were not classes. These eight primitives are: boolean, char, byte, short, int, long, float, and double. Apart from these eight data types, all other variables in a Java program must hold objects – or, more precisely, object references.

Objects are created or *instantiated* from classes. This concept is central to all object-oriented programming languages. The object represents a particular instance of the class. The state of this object is defined by the collective values of its internal data.

A key point for programmers new to OOP is the fact that the state of the object is *encapsulated* within that object. The programmer should not necessarily know the details of the internal state of the object or the internal behavior, which is defined by the code inside the methods in the class from which the object came. The programmer need only know the *interface* for this object.

The Application Programming Interface (API) for an object is usually composed of the methods from the object's class that have been marked as public. On rare occasions, a class designer may choose to mark a field as public, but this should be avoided – especially if the field is not a constant (i.e., final).

As previously mentioned, good class design is much more difficult than procedural (structured) program design. Therefore, students who have little or no experience with OOP should be provided with well-designed classes which employ solid, object-oriented design principles to use in creating an application-level program. In other words, it is our belief that students (especially those who have a background in structured programming techniques) should first be tasked with writing code that uses *existing* classes which make use of good encapsulation principles. Students can then experience how they should focus on the API and write their code to the interface and not to the implementation.

This is extremely important and is the central difference between object-oriented thinking and thinking in terms of procedures. The existing code that is made available to the students could be provided by the instructor, or it could come from the Java API. Even if you limit your students, as we do, to the java.lang and java.util packages, there are plenty of classes that can be used by the students in order to understand how to use the Java Class Library (via the Java API) just as effectively as code from the instructor.

Another important point to stress is that while the student may read the code in the classes provided by the instructor (hopefully to see an example of well-written code), this is not necessary in order to use the code. The code in the Java library is revealed by the Java API web pages – not source code. Well-designed classes are like black boxes. The students' only concern should be the API that provides the interface to the black box.

Once the students have gained confidence in the use of objects instantiated from existing classes provided by the Java Class Library and/or the instructor, they can then be tasked with the implementation of classes (given the design for those classes). This project represents one possible exercise that can help students make the move from procedural to object-oriented thinking.

THE "DEPARTMENT-EMPLOYEE" PROJECT

Project Background

The "Department-Employee" project was created to assess students' object-oriented programming skills with Java. This project asks students to implement a system that uses important object-oriented programming concepts and techniques. This includes classes and objects, aggregating classes and aggregated classes, encapsulation and information hiding, superclasses and subclasses, inheritance, method overriding and overloading, abstract classes and methods, polymorphism and dynamic binding. Specifically, this project asks students to implement four classes: Department, Employee, Consultant, and SalariedEmployee. There is an aggregation relationship between the Department class and the Employee class where Employee is the aggregating class and Department is the aggregated class. This indicates that an employee belongs to a department and that the department object is modeled as a field within the employee object. Both Consultant and SalariedEmployee are subclasses of the Employee base class.

Project Instructions

For this *Department-Employee* project, students are provided with four files: DepartmentEmployeeProjectInstructions.docx, DepartmentEmployeeUML.pdf, depts.txt, and DepartmentEmployeeTest.java. Students are required to use the UML class diagram in DepartmentEmployeeUML.pdf to implement the four classes: Department, Employee, Consultant, and SalariedEmployee. They are also required to use the provided DepartmentEmployeeTest.java to test their implementation. The correct output is also provided so that students can compare it with their program output for accuracy. When finished, students are required to zip all four .java files (Department.java, Employee.java, Consultant.java, and SalariedEmployee.java) and submit the archive file.

COURSE BACKGROUND, GRADING RUBRIC, AND SCORE REPORTS

The project was assigned to undergraduate Computer Information Systems (CIS) and Computer Science (CS) students enrolled in an advanced object-oriented programming course (comparable to a CS2 course). These students are generally juniors or seniors who are required to have completed an introductory programming course (comparable to a CS1 course) that focuses mainly on structured programming concepts, or an introduction to computer science course that focuses mainly on the theoretical foundations of computer science and the components of algorithms. The students are required to pass either of these two prerequisite courses with a grade of C or higher.

The content covered in this advanced object-oriented programming course can be roughly divided into five major groups: (1) fundamentals of programming, i.e., a quick review of structured programming such as control statements, methods, and arrays; (2) objected-oriented programming such as classes and objects, as well as inheritance and polymorphism; (3) GUI programming such as event-driven animations, UI controls, and multimedia; (4) data structures and algorithms such as lists, stacks, queues, binary search trees, and graphs; and (5) advanced Java programming such as multithreading, networking, and accessing databases with JDBC. The content of this course is comparable to the content suggested by Pendergast (2006). The "Department-Employee" project was assigned to the students after structured programming constructs were reviewed, object-oriented concepts were presented and practiced, but before the GUI programming techniques were introduced. Students had been given 10 previous less-complicated programming assignments before this one was introduced. These assignments focused on practicing with objects and classes, object-oriented thinking, inheritance and polymorphism, exception handling and text I/O, as well as abstract classes and interfaces. The students were required to complete the "Department-Employee" project within 72 hours after it was made available online. Students could work in or outside of a computer lab.

To simplify the grading process and ensure consistency, a grading rubric was developed for the assignment. Each submission for the project was required to contain four .java files, i.e., Department.java, Employee.java, Consultant.java, and SalariedEmployee.java. A perfect submission was assigned a score of 100 points (pts), with each perfect .java file being assigned a score of 25 pts. Specifically, each of the four .java files was assessed using a scale of six possible ratings, including (1) perfect - 25 pts, (2) good - 20 pts, (3) satisfactory - 15 pts, (4) below average - 10 pts, (5) clear failure - 5 pts, and (6) no submission - 0 pts.

This advanced objected-oriented programming with Java course is typically offered twice a year: online during the summer term, and face-to-face during the fall term. The project was assigned to students and the students' project scores were collected during the following four semesters: summer 2016, fall 2016, summer 2017, and fall 2017. The number of enrolled students for the course was 19 in summer 2016, 17 in fall 2016, 10 in summer 2017, and 22 in fall 2017. For the total of 68 students over the past four semesters, the average project score was 87.21 pts (87.21%). The difference among the average score for each .java file was not significant: Department.java (21.62, 86.47%), Employee.java (21.99, 87.94%), Consultant (21.76, 87.06%), and SalariedEmployee (21.84, 87.35%).

For the majority of students who did not earn a perfect score on this project, the primary reason was because they failed to correctly implement the toString() methods and/or the equals() methods for some or all of the four required classes. Students were expected to implement the toString() method based on the example output provided to them. The implementation of the equals() method was based on a simple object-oriented programming convention. That convention, discussed in the classroom, is that two objects (of the same class) are deemed to be equal when all of their corresponding fields have the same values (i.e., each field is "equal"). The requirements in this area provide an opportunity for interesting classroom discussions regarding what make objects "equal" and how these decisions should be made. These discussions should be revisited when database programming using JDBC is presented since primary keys can greatly simplify the implementation of the equals() method.

STUDENT PERCEPTIONS AND BACKGROUNDS

To evaluate the students' backgrounds and their perceptions of the project, an online survey was created using SurveyMonkey.com. Using this tool helped ensure anonymity so that students would be more comfortable in sharing their thoughts and perceptions. The survey included nine questions: two essay questions, six multiple choice questions, and one short answer question. The survey link was distributed to students by course announcement through Canvas, the university's learning management system (LMS). Participation in the survey was voluntary and no extra credit was awarded as an incentive for students to take the survey. Of the 68 students enrolled over the past four semesters, 56 (82.35%) students completed the survey.

The first essay question asked whether the student liked the project and why they did or did not like it. In total, 42 students (75.00%) responded positively towards the project. Students who responded positively said that the project helped them to better understand the object-oriented side of programming and the concept of inheritance. Some of the positive comments included:

- "Yes, the project is a good way to demonstrate both inheritance and aggregation in Java in one task. Overall, it wasn't very difficult but focused on two important concepts and required you to figure them out to learn."
- "Yes, it brought home a lot of Java concepts that I was still unsure of and reminded me how flexible object-oriented programming is."
- "I felt as if this project resembled more closely the programming techniques we would use at a real job such as file I/O, inheritance hierarchies, and exception handling."

Seven students (12.50%) responded negatively to the project, and 7 students (12.50%) responded neutrally to the project. The most commonly cited issue in the negative and neutral comments was a lack of clarity in the project instructions.

The second essay question asked students to list the most difficult challenges that they faced while working on the project. The challenges most commonly cited by students were the implementation of the toString() method and the equals() method. This is consistent with the instructor score reports. Other common challenges included vague instructions (as noted in the first essay question) and understanding when functions needed to be overridden. Some examples of these comments are:

- "The hardest challenge for me was the toString and equals method. I was confused on what I was supposed to be returning on each part of the code. After looking over the instructions a little more carefully I started to understand what parts were supposed to return."
- "My most difficult challenge was deciding what needed to go into the overridden equals in each file. We were only instructed on what we needed for the department file, but told to override all of the equals methods and I personally like to know more detail in my instructions. Other than that I did not have much of a problem with the assignment."
- "I still have trouble overriding the toString() and equals() methods. I can never remember which checks to use in the equals method, and it takes me a second to understand what exactly the toString() method is needing to return."

The remainder of the questions (three through nine) asked students questions relating to their backgrounds. In regards to question three, 37 students (66.07%) identified their major as Computer Information Systems, 17 students (30.36%) as Computer Science, and the remaining two students (3.57%) as Geographic Information Science. The results to question four indicated that 49 students (87.50%) were male, and 7 (12.50%) were female. Question five indicated that 47 students (83.93%) were seniors, 6 (10.71%) were juniors, 2 (3.57%) were sophomores, and 1 (1.79%) was of another classification. According to question six, 51 students (91.07%) were full-time students and 5 (8.93%) were part-time.

Question seven was a short answer question asking for the student's age. The results showed that the average age of the students surveyed was 23 with a standard deviation of 3.41. The highest recorded age was 37 and the lowest was 18. Question eight asked for the student's current overall GPA: 15 students (26.79%) replied with 3.50 – 4.00, 21 students (37.50%) responded with 3.00 – 3.49, 15 students (26.79%) replied with 2.50 – 2.99, and the remaining 5 students (8.93%)

responded with 2.00 – 2.49. The final question asked whether students had taken the course CIS 225: Introduction to Object Oriented Programming. Thirty-eight students (67.86%) said that they had taken the course and 18 students (32.14%) said that they had not.

CONCLUSIONS

This article described the "Department-Employee" project, which was used to assess students' object-oriented programming skills with Java. The article reviewed the history of object-oriented programming and provided an overview of object-oriented programming with Java. It provided the project specification as well as the course background, grading rubric, and score reports. It also presented survey data for evaluating students' backgrounds, as well as the students' perceptions regarding the project. Results from the instructor score reports and student perceptions showed that the project was effective for assessing students' object-oriented programming skills with Java.

Teaching object-oriented programming and helping students think in terms of objects and classes can be a challenge. The project presented in this paper, along with the guidelines and suggestions, should assist instructors in moving students toward that goal. We have demonstrated that our students have benefited from this assignment and we endeavor to find or create similar tools that help us gauge student comprehension of object-oriented programming concepts.

REFERENCES

1. Benander, A. C., Benander, B. A., & Lin, M. (2003) Perceptions of Java - Experienced programmers' perspective. *Journal of Computer Information Systems*, 43, 4, 1-7.
2. Booch, G. (1991) *Object-oriented analysis and design with applications*. Cambridge, MA, USA: Addison-Wesley.
3. Cavaiani, T. P. (2006) Object-oriented programming principles and the Java class library. *Journal of Information Systems Education*, 17, 4, 365-368.
4. Eckel, B. (1998) *Thinking in Java* (1st edition). Upper Saddle River, NJ, USA: Prentice Hall PTR.
5. Hu, C. (2008) Just say 'a class defines a data type.' *Communications of the ACM*, 51, 3, 19-21.
6. Jacobson, I. (1993) *Object-oriented software engineering: A use case driven approach*. Cambridge, MA, USA: Addison-Wesley.
7. Johnson, R. A., & Moses, D. R. (2008) Objects-first vs. structures-first approaches to OO programming education: An empirical study. *Academy of Information and Management Sciences Journal*, 11, 2, 95-102.
8. Meyerovich, L. A., & Rabkin, A. S. (2013) Empirical analysis of programming language adoption. *ACM SIGPLAN Notices*, 48, 10, 1-18.
9. Ourosoff, N. (2002) Primitive types in Java considered harmful. *Communications of the ACM*, 45, 8, 105-106.
10. Pendergast, M. O. (2006) Teaching introductory programming to IS students: Java problems and pitfalls. *Journal of Information Technology Education*, 5, 491-515.
11. Pokkunuri, B. P. (1989) Object oriented programming. *ACM SIGPLAN Notices*, 24, 11, 96-101.
12. Ramesh, V., & Wu, J.-T. B. (2004) An approach to teaching object-oriented programming concepts in business schools. *Decision Sciences Journal of Innovative Education*, 2, 1, 83-87.
13. Rentsch, T. (1982) Object oriented programming. *ACM SIGPLAN Notices*, 17, 9, 51-57.
14. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991) *Object-oriented modeling and design*. Englewood Cliffs, NJ, USA: Prentice-Hall.
15. Shein, E. (2015) Python for beginners. *Communications of the ACM*, 58, 3, 19-21.
16. *TIOBE index*. (2018) Retrieved from https://www.tiobe.com/tiobe-index/
17. Weisfeld, M. (2008) *The object-oriented thought process* (3rd edition). Cambridge, MA, USA: Addison-Wesley.
18. Westfall, R. (2001) Hello, world considered harmful. *Communications of the ACM*, 44, 10, 129-130.
19. Xing, C.-C., & Belkhouche, B. (2003) On pseudo object-oriented programming considered harmful. *Communications of the ACM*, 46, 10, 115-117.
20. Zhang, X. (2010) Assessing students' structured programming skills with Java: The "blue, berry, and blueberry" assignment. *Journal of Information Technology Education: Innovations in Practice*, 9, 227-235.