Beginners Understanding of Object Oriented Programming

Pelagia Majoni

19 November 2021

1 Abstract

Problems beginners face with understanding Object Oriented Programming (OOP) concepts have been investigated and reported in a number of studies. In this paper, we present the results of comparing and contrasting the methods and results of eight studies on novice’s misconceptions of OOP. We identify the types of assessment each study used the teaching methods researchers used and finally the types of content students in different studies were expected to produce and compared and contrasted the methodology and results of each of the researchers. In these studies, students were asked to draw diagrams depicting the notional machine. Researchers defined systematic ways to analyze student’s drawings for OOP misconceptions. The main findings highlighted that a significant number of students fail to distinguish between classes and objects.
## Contents

1 Abstract .................................................. 1

2 Introduction ........................................... 3

3 Background Information .................................. 3
   3.1 Building Blocks of OOP ................................. 4
   3.2 Principles of OOP ....................................... 4
   3.3 OOP Languages ......................................... 5
   3.4 Benefits of OOP ........................................ 5

4 Motivation ................................................. 5

5 Literature Review ......................................... 6
   5.1 Methods .................................................. 6
      5.1.1 Model Classes Design Focused Studies .......... 6
      5.1.2 Programming and Code Focused Studies .......... 8
      5.1.3 Theoretical Knowledge Focused Studies .......... 8
      5.1.4 Notional Machine Focused Studies ............... 9
   5.2 Results .................................................. 14
      5.2.1 Instance/class conflation ......................... 14
      5.2.2 Failures in modelling .............................. 14
      5.2.3 Dynamic View of Notional Machines ............... 16
      5.2.4 Class Reference/ Non-reference ................... 16
      5.2.5 Problems with hierarchies and abstraction ....... 16
      5.2.6 Overall Takeaways .................................. 18
      5.2.7 Future Work ........................................ 19
2 Introduction

Students’ understanding of OOP is an important topic that has been studied in the field of Computer Science education to help students understand OOP more easily. In this paper, we analyzed those studies on beginners’ understanding of OOP. We look at the different methods researchers used and this included oral interviews, transcripts, questionnaires, written exams. Students were orally interviewed on OOP concepts and they were also given midterm exams. This was after students had taken an Introduction to CS class that also taught OOP. In this paper, we also included the topics included in the students’ CS curriculum, examples of questions were asked and the tasks that they were give. Students were asked by researchers to produce drawings, concept maps, text to depict their understanding of OOP.

After systematically analyzing students’ work, researchers reported the main types of misconceptions students have of OOP concepts. Students’ mental visualization of the program state where shown to not only evolve but to change with time and with the introduction of new tools. They also found that their students seem to be experiencing code as no more than text that is merely copied and pasted rather than something with real world meaning. Lastly, there was a common fault of students creating classes that they never referenced.

In this paper, we first introduce the reader to key OOP concepts, and then we discuss the methods researchers used and their findings. Research had some similar findings and these were grouped into the same categories in this paper. We finally have a future work section proposing to investigate students’ understanding of when to and when not to use OOP in given programming tasks.

3 Background Information

What is OOP and why is it important? OOP is a computer programming technique that organizes software design around data, or objects, instead of functions and logic. In OOP, a class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods). The user-defined objects are created using the class keyword. The first step in OOP is to collect all of the objects a programmer wants to manipulate and identify how they relate to each other - an exercise known as data modeling. Once an object is known, it is labeled with a class of objects that defines the kind of data it contains and logical sequences methods that can manipulate it. Objects can communicate with each other or call each other (i.e., object messaging).
3.1 Building Blocks of OOP

The structure, or building blocks, of object-oriented programming include the following:

- **Classes** are user-defined data types that act as the blueprint for individual objects, attributes and methods.

- **Objects** are instances of a class created with specifically defined data. Objects can correspond to real-world objects or an abstract entity.

- **Methods** are functions that are defined inside a class that describe the behaviors of an object. Each method contained in class definitions starts with a reference to an instance object. Programmers use methods for re-usability or keeping functionality encapsulated inside one object at a time.

- **Attributes** are defined in the class template and represent the state of an object. Objects will have data stored in the attributes field. Class attributes belong to the class itself.

3.2 Principles of OOP

Object-oriented programming is based on the following principles:

- **Encapsulation** This principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.

- **Abstraction** Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. The derived class can have its functionality extended. This concept can help developers more easily make additional changes or additions over time.

- **Inheritance** Classes can reuse code from other classes. Relationships and sub-classes between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy. This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.

- **Polymorphism** Objects are designed to share behaviors and they can take on more than one form. The program will determine which meaning or usage is necessary for each execution of that object from a parent class, reducing the need to duplicate code. A child class is then created,
which extends the functionality of the parent class. Polymorphism allows different types of objects to pass through the same interface.

3.3 OOP Languages

Pure popular OOP languages include: Ruby, Scala, JADE, Simula and Emerald. Programming languages designed primarily for OOP include: Java, Python and C++. Other programming languages that pair with OOP include: Visual Basic .NET, PHP, JavaScript.

3.4 Benefits of OOP

The benefits of OOP include: modularity, re-usability, productivity, scalability and efficiency. OOP focuses on the objects that programmers want to manipulate instead of the logic required to manipulate them. This technique works very well for complex and large projects that require constant updates and maintenance. Examples of such programs include operating systems, manufacturing and design, as well as mobile applications. In addition to scalability, the organization of an object-oriented program also makes the OOP a good tool for collaborative software development, where projects are subdivided into groups.

While OOP modularity, re-usability, productivity, scalability and efficiency, it has its downfalls. OOP code may be more complicated to write and may take longer to compile. OOP is difficult for beginners to understand and, as such, is the motivation for doing research on beginners comprehension of OOP.

4 Motivation

OOP is an important technique for students learning Computer Science as it allows programmers to write readable and efficient code. It is easier to think in object oriented terms, because it is similar to how the object being modeled happens in the real world. Functional programming is all about data manipulation. Converting a real world scenario to just data can take some extra thinking.

OOP is one core tool that allows programmers to break the program into the bit-sized problems that can be solved one object at a time. OOP also enables developers to build programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity. Unfortunately, most beginners struggle to understand the main concepts behind OOP. Learning OOP is a cognitively demanding challenge for beginner programmers [Xin15]. Given the importance of OOP education, this paper compares and contrasts studies on novices’ comprehension of OOP concepts with the future goal of conducting a study to understand beginners’ ability to discern when not and when to use OOP.
5 Literature Review

5.1 Methods

5.1.1 Model Classes Design Focused Studies

Thomasson et al. and Sanders et al. studies focused on analyzing students’ class designs to understand problems novice programmers face when learning OOP. Thomasson et al.’s [TRT06] students were instructed to draw class designs for problems similar to the ones produced in class, lab and tutorials. The problems assigned to the students were specifically designed to investigate the level at which students had understood the process of breaking down a problem into its component parts. Thomasson et al.’s study focused on students’ class designs for a problem solution and required no representation of code. They only looked at class attributes as expressed by instance variables.

Thomasson et al. analyzed the designs of 180 students, 115 were novices and 65 had some programming experience. Students were asked to model a paper round system consisting of a person delivering orders to customers. The students were told to identify the classes required, their attributes and their methods. Students drew their designs on paper, and the researchers manually analyzed their classes. In the second phase, the same students were given a different problem, in which they were required to design classes for a car-hire company with multiple depots, cars and customers. The students were again asked only to produce class diagrams to support the final system. In the last phase, conducted the following year, a different group of 90 students was given the car-hire problem.

Similar to Thomasson et al., Sanders et al.’s [SBE+08] study focused on students’ design of model classes. Sanders et al. first gave students a sample concept map and then collected 119 concept maps from 107 participants. Some students were asked to created more than one map. The sample concept map is show below. The participants included 71 novices (first-year students who had been introduced to encapsulation, inheritance, and polymorphism), 12 intermediate students, 15 graduating students, and 9 instructors. All students were given a sample concept map that started with the concepts “kitchen” and “dinner” and the following problem: “Put the concept map here that starts with the two concepts “class” and “instance” with labeled arrows and other concepts that creates a partial map of object-oriented programming, as you have learned it so far.”
The researchers normalized the data, replacing labels like “contain,” “contains,” and “may contain” with a uniform “contains.” They computed the most common node names, the most common edge labels, and the most frequent “sentences.” The researchers agreed on what might indicate an implicit reference, closely examined the maps, and then compared their results.

Sanders et al. and Thomasson et al. both looked at student’s designs but they used different methods to analyze them. Unlike Thomasson et al., Sanders et al. were more interested in comparing students’ maps to each other than to their model answer.
5.1.2 Programming and Code Focused Studies

Sanders and Thomas and Holland et al.’s studies focused on looking at students’ code and programs. Sanders and Thomas [ST07] gave students five different programming assignments, each focusing on classes, inheritance and polymorphism. Holland et al. [HGW97] offer suggestions for how to avoid fostering misconceptions about Object-Oriented concepts in novices. Holland et al.’s suggestions are based on their experience of designing and teaching introductory courses rather than on an empirical analysis of their student’s programs.

Sanders and Thomas carried out a close examination of programs written by 16 students in an objects-first CS1 course. Five of the students had previous programming backgrounds. Students were given 5 programming projects, all of which were manually examined by the researchers for misconceptions. Similar to Holland et al., the assigned projects were based on concepts previously covered in lectures and in the lab. In total, the students submitted 71 programs.

In Program 1 the students were to: demonstrate understanding of defining classes, instance variables, and methods; show understanding of the containment (has-a) hierarchy by defining and instantiating a composite class; show an ability to work with constructors and parameters; and show an understanding of the class-instance distinction by instantiating three different objects using the same class. Program 2 required students to create objects that interact with each other and to create a small inheritance hierarchy of classes, where instances of the sub-classes would behave polymorphically. For program 3, the goal was for students to demonstrate a deeper understanding of interacting objects and abstraction by adapting and using a simple design pattern, consisting of one object that manages a particular piece of data and two or more objects that use that data. In Program 4 there was an opportunity to define an inheritance hierarchy (by defining a blue-print class and extending it for different shapes). Whether or not the students chose to do so (with no prompt) gave the researchers some sense of how well they had integrated inheritance into their mental model. Finally, Program 5 was intended to be a capstone experience for the semester, pulling together the concepts previously studied. Sanders and Thomas’s study focused on the details of students’ code.

5.1.3 Theoretical Knowledge Focused Studies

Eckerdal and Thune, Ragonis and Ben-Ari, and Stelios Xinogalo’s studies asked students open-ended theoretical questions on OOP concepts to gauge students’ understanding of OOP. Eckerdal and Thune’s [ET05] study asked students theoretical questions about OOP in an oral interview while Stelios Xinogalo’s [Xin15] study asked students theoretical OOP questions in a written-format midterm exam.

Eckerdal and Thune interviewed 14 first year students on their understanding of the concepts of object and class. The students had just taken an introductory Computer Science course. The interviews were intentionally unstructured and encouraged students to demonstrate as much of their knowledge on the question
asked as possible. The interviewers transcribed and analyzed the students’ responses.

Eckerdal and Thune were not aiming for statistically significant results. The objective in selecting persons to interview was to get as broad a coverage as possible of different conceptions. Two researchers independently read and analyzed the interviews, looking for qualitatively different ways to understand the concepts object and class expressed in the data. This study, like Thomasson et al.’s and Holland et al.’s focused on the basic OOP concepts of objects and classes and did not include more advanced concepts such as inheritance and polymorphism.

Stelios Xinogalo carried out a long term study with the same group of students. The researcher collected data in the context of mid-term exams. Open-type questions were included in the exam for investigating students’ conceptions of the notions of class. Fifty-six students participated in the exams and consequently in this study.

During two academic years, Ragonis and Ben-Ari [RBA05] taught OOP to high school students, using Java and BlueJ. They specifically focused on the following concepts: class vs. object, instantiation and constructors. Extensive data collection was carried out through the entire period: observations and field notes, audio and video recordings, and collection of artifacts. Ragonis and Ben-Ari used homework assignments, class work, examinations, and a final project to determine what concepts students understood.

5.1.4 Notional Machine Focused Studies

With the goal of studying the development of students’ visualization of OO program state, Sajaniemi et al. [SKT08] did their research on a group of students who were taking an introductory programming course. Below is a table summarizing the material covered in the course the students were taking. The course used pictures and animation tools to visually display OOP concepts and program execution to their students. Sajaniemi et al. were interested in understanding the effectiveness of using such tools. They wanted to see their impact on students’ notional machines. In order to study students’ own visualizations of OO program state (i.e., the notional OO machine) and their (mis)conceptions of OO concepts, Sajaniemi et al. instructed students to draw pictures depicting program state at a specific point of program execution. The researchers gave students a short program and asked them to draw a picture depicting the state of the program at any given point of execution. The students were specifically asked to draw a picture that includes existing objects and methods and their relationship at specific points of the program shown in Fig. 2.
Students were given minimal instructions in order for the researchers to see what students considered to be the key concepts in an OO program. As the semester progressed, students were exposed to different tools to visualize their programs and they were given 2 more similar tasks. The Java program Sajaniemi et al. used in the second and third drawing tasks (translated from Finnish). Students’ drawings were then analyzed by looking at the types of elements they contained and the relationships between these elements. Their analysis looked at three aspects of a drawing: (1) impact of previously seen visualizations, (2) form of the drawing, and (3) the contents of the drawing. One of the students’ drawings is shown below (Fig. 3) and we discuss this students’ and other students results in the findings section.
public class Town {
    public static void main (String[] args) {
        Owner owner_a = new Owner("John",50);
        Owner owner_b = new Owner("Pete",20);
        Pet dog1 = new Pet("Bucky", owner_a);
        Pet dog2 = new Pet("Boo", owner_a);
        Pet dog3 = new Pet("Tooty", owner_b);
        owner_b.increaseFood(100);
        dog3.hunger(5); // --- During this method call
dog3.hunger(4); //
    }
}

// --------------------------------

class Pet {
    private String name;
    private Owner owner;
    public Pet (String n, Owner o) {
        name = n; owner = o;
    }
    public void hunger (int wish) {
        int given;
        given = owner.feed(wish);
        System.out.println(name + " has eaten " + given);
    }
}

// --------------------------------

class Owner {
    private String firstName;
    private int food;
    public Owner (String n, int f) {
        firstName = n; food = f;
    }
    public int feed (int wished) {
        int portion = wished;
        if (wished > food) portion = food;
        food = food - portion;
        // --- at this point
        return(portion);
    }
    public void increaseFood (int increment) {
        food = food + increment;
    }
}

[SKT08] Fig. 2. The Java program used in the second and third drawing
tasks (translated from Finnish).
Ford [For93] as cited in [SKT08] researched a group of students that had been taught OO design and programming for one semester. Ford asked teams made of 2 to 4 students to design and implement animations that exemplified features of the C++ programming language. Ford used exploratory data collection followed by interviews with the students. Ford focused on dynamics involving variables and classes and noted that these show how students visualized the notional machine behind the language.

Students’ visualizations were analyzed in an exploratory way and the only portion of their results that were reported, were presented with the goal of illustrating the variety of student visualizations. Visualizations of OO concepts were treated neither systematically nor extensively. This study is closest to Sajaniemi et al.’s study, but Sajaniemi et al.’s study did the following differently: they used pen and paper instead of animation as they found animations to be too laborious, their students gave a static visualization of program state instead of a dynamic animation, their students worked alone and not in groups, they gathered data at several points during a programming course, and they analyzed students’ visualizations systematically and in detail rather than examining them to demonstrate the diversity of students’ visualizations.

Teif and Hazzan [TH06] as cited in [SKT08] studied junior high school students’ interpretation of class and object using questionnaires, interviews, videotapes, and a researcher diary. The questionnaires included drawing tasks where students were asked to draw the class-object relation; later, students were interviewed and could explain their drawings. Teif and Hazzan’s students’ drawings and explanations revealed several misconceptions (object-class relations confusion). In this study, students’ drawings do not depict a specific program or program state but are pictures of everyday things intended to visualize class-object relation at a more general level; students’ drawings are analyzed for OO misconceptions.
Thomas et al. studied first-year programming students’ understanding of program behavior and especially of object referencing. They used object diagrams in many examples when tracing code during class. They then conducted an examination consisting of multiple-choice questions on program tracing. In the test, half of the students were given ready-made partial object diagrams. Several weeks later a new test was conducted with no ready-made diagrams, and students’ scratch sheets were collected and analyzed for their use of object diagrams. Thomas et al. found that students did not perform better in tracing OO code fragments when they were provided with the ready-made diagrams, nor did they draw their own diagrams more often in the follow-up test. Thomas et al. concluded that the inability to use diagrams in program tracing is a widespread phenomenon among beginning, weaker students.
5.2 Results

5.2.1 Instance/class conflation

Sanders and Thomas, Holand et al., Ragonis and Ben-Ari, and Sanders et al.’s studies observed that some students failed to differentiate between an instance of a class and a class. While Teif and Haazan’s study observed that students think of instances as a subset of a class, Xinogalos found that most students did not (except for 1 out of their 14 participants.) This was specifically true for user defined classes and not built in library classes.

Sajaniemi et al. found that students did not clearly see the difference between classes and objects. This is because students depicted currently active program code within the nested method calls with no reference to the associated objects. Students mentioned objects solely as a part of the active program code only. Thus students did not think of objects as important in describing program state, they thought of classes and program code as important but left out objects. Thus here we see object-class conflation.

Sanders and Thomas observed several problems distinguishing between user-defined classes and instances of those classes. Two students, for example, defined separate classes for Program 1 and instantiated each one once, instead of instantiating the same class three times (e.g., Head, Head 1, and Head 2; Body, Body 1, and Body 2). The classes have the same fields and methods; they differ only in their property values.

Some of Sanders et al.’s students describe an instance as being the same thing as a class while some said an instance is an example of a class. Some students, however, correctly understood that a class is a blueprint for creating instances or objects.

Ragonis’s study also found that students did not necessarily view a class as a template. Once they called a class once, they did not call it again even when they had to. They also did the same for class methods.

Holland et al. also observed that students sometimes showed signs of confusion when it came to the difference between an object and a class. They sometimes referred to these two as one thing. Holland et al. suggested that instructors ensure that they create multiple instances from one class, so that students do not conflate the two.

5.2.2 Failures in modelling

Sanders and Thomas and Eckerdal and Thunes’ study found that their students appear to be experiencing code as no more than text that is merely copied and pasted rather than something with real world meaning. Their students’, similar to Eckerdal and Thunes’, created objects that functioned within a program, but did not correspond to the domain.

When asked to describe a class, a couple of students expressed something along the lines of student X who said: “A class is, well I figure a class is like a small program, that’s how I think of it, a small program inside the whole big program, if you say that the big program is the main program, then the class is
like a small program doing certain things.” This group of students focus on the
program structure and the programmers task and describes the class-concept as
a help for the programmer when structuring the code.

Sanders et al also found that most of their students thought of classes as
modelling real world phenomena. According to Xinogalos’ study, few students
think of “classes” merely as entities in the program, while more students think
of “objects” as pieces of code as shown in the table below. Overall, Xinogalos’,
Sanders and Thomas’s and Eckerdal and Thunes’ studies all found that there
is quite a number of students who think of classes as merely code in the program.

<table>
<thead>
<tr>
<th>Conceptions</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Contains data and methods</td>
<td>12.5%</td>
</tr>
<tr>
<td>Piece of code</td>
<td>5.36%</td>
</tr>
<tr>
<td>An active entity (dynamic nature)</td>
<td>19.0%</td>
</tr>
<tr>
<td>An object is an instance of a class/s constructed from classes</td>
<td>3.57%</td>
</tr>
<tr>
<td>Something that is active in the program</td>
<td>5.36%</td>
</tr>
<tr>
<td>An object is, for example, a robot that is constructed in the main program in order to execute the statements</td>
<td>5.36%</td>
</tr>
<tr>
<td>An object executes whatever you tell it to execute with various statements</td>
<td>3.57%</td>
</tr>
<tr>
<td>Objects execute statements with the help of classes that define their kind</td>
<td>3.57%</td>
</tr>
<tr>
<td>Model of some real-world phenomena</td>
<td>50%</td>
</tr>
<tr>
<td>A model (entity) of some real-world phenomenon</td>
<td>3.57%</td>
</tr>
<tr>
<td>An object is something concrete in contrast with a class that is a more general/abstract concept</td>
<td>3.57%</td>
</tr>
<tr>
<td>Entity in the program contributing to the structure of the code</td>
<td>3.57%</td>
</tr>
<tr>
<td>Description of properties and behavior of the object</td>
<td>3.57%</td>
</tr>
<tr>
<td>An object is something concrete in contrast with a class that is more general - objects are a subset of classes</td>
<td>3.57%</td>
</tr>
<tr>
<td>Model describing a kind of object</td>
<td>69.64%</td>
</tr>
<tr>
<td>In a class we define what an object does/the statements that are going to be executed are contained</td>
<td>7.14%</td>
</tr>
</tbody>
</table>

[Xin15] Fig. 3. The Java program used in the second and third drawing
tasks (translated from Finnish).
5.2.3 Dynamic View of Notional Machines

Sajaniemi et al. found that students’ mental representations of OO concepts and program execution not only evolve as new material is covered in teaching, but they also change. The first set of drawings they had students do treat methods as having primarily a static existence. In the next set of drawings, methods are seen as dynamic invocations that call each other. The role of classes in program execution fluctuated during learning, indicating problems in understanding the relationship between classes and their instances. They also found that students struggled to understand the relationship between an object and methods. In the second task, Sajaniemi et al. asked the student to describe in his or her own words the concept of object. Below is one of their students’ drawings.

The drawing contained the following errors: the name of the first attribute in “Owner” objects should be “firstName” (instead of “name”); the updated value of the attribute “owner.b.food” should be 120 (instead of 100); the arrow labeled “100” should not start from “owner a” (but from the “main” method not depicted in the drawing); and the arrow labeled “hunger(5)” should not start from “dog1” (but from the “main” method, also). The researchers however scored Fig. 3 as correct because the drawing contains mainly correct attribute names and they were not interested in calculation errors.

Now that we have zoomed in on how they analyzed one of the students’ drawings, we have an idea of how they analyzed the other 40 students’ drawings. The researchers found that the existence or execution of the “main” method was not depicted in any drawing. From this, Sajaniemi et al. concluded that students’ mental representation of the “main” method is so unclear that they fail to depict it in a drawing at all.

5.2.4 Class Reference/ Non-reference

Analysis of the students’ designs revealed that most students developed a class in isolation and failed to utilise it within their proposed system. Novices understood the necessity for a concept, but were unable to relate it to other classes through appropriate use of instance variables.

Sanders and Thomas observed that as the semester progressed, students occasionally referenced nonexistent classes. They argue that this error might have been due to the increased complexity of the programs, rather than misconceptions about linking.

5.2.5 Problems with hierarchies and abstraction

Sanders and Thomas found that their students did not always use inheritance correctly. In some instances, what students portrayed as sub-classes should have been instances of their defined classes. Sanders and Thomas’ students, in multiple cases, failed to use inheritance when the problem could have been better solved through the use of inheritance. Some students, for example, defined separate classes for objects that were supposed to inherit from one parent class.
Sanders and Thomas argue that the students made these errors because it’s simple to copy and paste code from one class to another and there’s no immediate incentive to factor out code. Sanders and Thomas summarized their results in two checklists (Tables 2 and Table 3) of things instructors can look for when grading introductory Computer Science courses’ programs with the hope that these checklists will help professors both in designing assignments that test particular concepts and misconceptions and in grading those assignments.

<table>
<thead>
<tr>
<th>Some things to look for</th>
<th>Suggests understanding of:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program complex</td>
<td>Basic mechanics</td>
</tr>
<tr>
<td>Constructors defined and used</td>
<td>Constructors</td>
</tr>
<tr>
<td>Multiple instances of same class</td>
<td>Object and class</td>
</tr>
<tr>
<td>Variables / methods with same name in different classes</td>
<td>Encapsulation</td>
</tr>
<tr>
<td>Multiple classes defined in program</td>
<td>Linking, message passing methods</td>
</tr>
<tr>
<td>Composite object (object with parts) defined</td>
<td></td>
</tr>
<tr>
<td>Object passed as parameter to constructor (peer object)</td>
<td></td>
</tr>
<tr>
<td>Peer object assigned to instance variable</td>
<td></td>
</tr>
<tr>
<td>Methods other than constructors defined</td>
<td></td>
</tr>
<tr>
<td>Message sent to part / peer object</td>
<td></td>
</tr>
<tr>
<td>Methods’ return values used</td>
<td></td>
</tr>
<tr>
<td>Library classes extended</td>
<td>Inheritance</td>
</tr>
<tr>
<td>User-defined classes extended</td>
<td></td>
</tr>
<tr>
<td>Shared properties/methods factored into superclass</td>
<td></td>
</tr>
<tr>
<td>Single library class has multiple subclasses that define the same method differently</td>
<td>Polymorphism</td>
</tr>
<tr>
<td>Single user-defined class has multiple subclasses that define the same method differently</td>
<td></td>
</tr>
<tr>
<td>Classes correspond to visible objects in domain</td>
<td>Modelling</td>
</tr>
<tr>
<td>Some class corresponds to invisible (conceptual) object</td>
<td></td>
</tr>
<tr>
<td>Methods have parameters</td>
<td>Abstraction</td>
</tr>
<tr>
<td>Method parameters used</td>
<td></td>
</tr>
<tr>
<td>Inheritance hierarchies defined and used</td>
<td></td>
</tr>
<tr>
<td>Simple recipes used</td>
<td></td>
</tr>
<tr>
<td>Design patterns used</td>
<td></td>
</tr>
</tbody>
</table>

[ST07] Table. 2. Indications that a student understands basic OO concepts.
Overall Takeaways

Overall, students in most of these studies referenced above showed two consistent misunderstandings: failure to distinguish between a class and its instance and they also a significant number of them did not think of classes as modeling real world entities. In addition to these errors, notional machine focused studies found that students struggle to link methods with instances. Students mental visualization of the program state where shown to not only evolve but to change with time and with the introduction of new tools. They also found that their students seem to be experiencing code as no more than text that is merely copied and pasted rather than something with real world meaning. Lastly, there was a common fault of students creating classes that they never referenced.
5.2.7 Future Work

One area we are interested in exploring and studying further is beginners’ understanding of when to and when not to use OOP tools such as classes and inheritance. Research on beginners’ understanding of OOP has revealed several misconceptions and problems that concern the notion of class and object. While a lot of work has been done on understanding the difficulties novice programmers face when learning OOP, there is not much literature on beginners’ knowledge of when and when not to use OOP in their projects. In our proposed study, we will give Computer Science students in their second semester of college two assignments, one that requires them to use OOP classes but does not tell them to, and one that does not need OOP classes. This research question would be worth exploring as it will contribute to the growing body of literature on OOP introductory education.

References


