A Novel Approach to Introduce Classes in C++

Lasanthi Gamage
Mathematics and Computer Science Department
Webster University
St. Louis, MO 63110
lasanthigamage67@webster.edu

Jayantha Herath
Department of Computer Science
St. Cloud State University
St. Cloud, MN 56304
jherath@stcloudstate.edu

***Abstract*—This article discusses the approach of using UML to introduce classes for the first-time Object-oriented programming takers; it is worth noting here that the students are not expected to have prior experience in UML. The discussion presented in this article includes a collection of example UML diagrams that are used to demonstrate some selected concepts of Object-oriented programming, including, class definitions, class functions, access modifiers, constructors, and inheritance. This method of introducing object-oriented concepts has put into practice for several semesters and has attracted students' gratification.**

I. Introduction

When it comes to object-oriented programming, many students struggle to grasp the concepts. This is not necessarily because of students' lack of logical thinking; even students who were very good with procedural programming, sometimes fail to understand Object-oriented programming (OOP) concepts because the differences between the two are not clearly and explicitly explained.

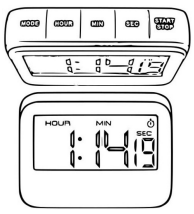
Most pedagogical approaches introduce programming concepts to students one at a time, starting from, for example, the general structure of the class, the class definition, member functions, access specifiers, constructors, and so on. This step-by-step build-up of concepts naturally takes at least a couple of lectures before the students can start to develop a solid understanding of classes and class interactions. Such an approach, in many ways, is akin to a bottom-up introduction to the TCP/IP stack (in a networking class) where the physical layer is introduced first and the application layer is introduced the last. Alternatively, it is also possible to first start with classes and their interactions directly, and then deep dive into the associated details. This is similar, in nature, to a top-down approach.

Visual representations of classes using UML-like visual aids are an effective way to jump-start an introduction to object-oriented programming concepts. Instead of getting bogged down on details at the onset, UML help students *see the forest through the trees* and grasp the benefits of core OOP concepts such as abstraction, aggregation, encapsulation, and association much faster. Furthermore, UML diagrams can be easily converted into classes fast-tracking actual programming.

The rest of the paper is organized as follows. Section I discusses how to introduce classes using UML. Section II discusses how UML can be then used to introduce class instances (objects) and constructors. Section III discusses the techniques used to introduce function overloading using constructors as a case study. Section IV discusses how UML is effective in introducing class hierarchies and Inheritance. Section V gives a summary of work related to teaching introductory level programming. Lastly, a conclusion of the presented work is provided in Section VI.

Introduction to Classes

A good choice of topic to motivate students on the advantage of Classes is *parallel arrays*. Many textbooks readily discuss parallel arrays in some contexts, making it an excellent and easily adaptable topic for instructors. For example, part of the chapter on Arrays in Gaddis' textbook is dedicated to parallel arrays [1]. Savitch, in his book, does not use the exact term, but introduces the concept as part of a discussion on a 2-D grading program [2].



Clock

- hour : int
- min : int
- sec : int

+ setTime(int, int, int) : void
+ getTime(int&, int&, int&) const: void
+ printTime() const : void
+ incrementSecs() : void
+ incrementMins() : void
+ incrementHours() : void
+ equalTime(const Clock&) const : bool

Fig. 1: UML Class Diagram for a Clock

Parallel arrays can be used to keep track of different attributes of a list of objects and provide plenty of easily tangible (real-world) case studies. Two of which are explained herewith. First, consider students in a classroom. Each student could be tracked for several attributes such as name (*string*), id (*int*), average quiz score (*double*), seating index (*short*), etc. Without the use of objects (or Classes), each attribute will have to be maintained in a separate array of its own because of the data type differences and because each attribute explain a different feature of a student. It's not difficult to explain

how tracking attributes in this manner becomes cumbersome and complex for large classrooms, or across several classes over a span of semesters. In fact, the complexity of tracking attributes in this manner can be a great discussion point on the advantages of **aggregation**, where a single "*custom variable*", i.e. a single Student object, can aggregate multiple attributes into one.

Second, consider a Clock, more specifically, a Digital Clock. Limiting ourselves to only functional attributes[1], what are the attributes associated with a digital clock? Consider Figure 1 which describes a clock object and several ways to interact with it in a UML diagram. This UML diagram has three parts: (i) the name of the object; (ii) three integer attributes to denote hours, minutes, and seconds; and (iii) several methods /functions that are used to interact (or to communicate) with the clock. Each attribute is identified by an access modifier (+ or - sign), an attribute name, and an attribute type separated from the attribute name using a colon (`:`). Similarly, methods/functions are identified by an access modifier, function signature (function name and a list of parameters), and the return type separated by a colon. The `(+)` access modifier indicates that the corresponding element is publicly accessible for the user. The `(-)` access modifier indicates that the element is hidden from or inaccessible to user interaction.

The UML diagram approach is a simple, direct, and intuitive way for students to understand aggregation. The combination of attributes, access modifiers, and functions help students distinguish between elements of an object that should be exposed for user manipulation and those that should not. Thus, students start to understand much deeper OOP concepts such as **abstraction** and **encapsulation** in a more direct and easily understandable manner. This understanding also helps students quickly translate the UML diagram into a programming code. For example, consider Listing 1 which depicts a direct translation of the UML diagram in Figure 1 to a C++ code.

[1]hence ignoring physical attributes like color, shape, etc.

```cpp
#include <iostream>
using namespace std;

class clock
{
    private:
        int hour;
        int min;
        int sec;

    public:
        void setTime(int, int, int);
        void getTime(int&, int&, int&) const;
        void printTime() const;
        void incrementSecs();
        void incrementMins();
        void incrementHours();
        bool equalTime(clock&);
}
```

Listing 1: C++ Translation of the clock UML Diagram

As shown, the C++ code is nearly a word-to-word copy of the UML diagram. Important points to highlight during instruction are to show that the plus sign (or the `(+)` access modifier) translates into **`public`** elements of the class in the programming code, while the minus sign (the `(-)` access modifier) translates into **`private`** elements. The name of the object in UML translates to the name of the class prefixed with the keyword **`class`**. At this level, no emphasis is placed on creating objects or instances of a class. The main focus here is on **Aggregation** using a class, along supplemental emphasis on **Abstraction** and **Encapsulation**. Further re-enforcement of these key OOP concepts can be achieved by providing students with additional UML examples that they are asked to translate into the corresponding programming code.

What's left this exercise at this point is for students to fill-in the details of each function definition. This can be assigned as an active learning exercise, an assignment, or a quiz. Based on classroom experience, students do not find such exercises overly difficult as they already possess necessary programming background from their introductory programming courses. There is also greater flexibility for the instructor to change function definitions between semesters or different sections of the same offering. As an example, Listing 2 depicts a sample implementation of the `incrementHours()` function in which students were instructed to implement a 24-hour format digital clock.

```cpp
void Clock::incrementHours()
{
    // assuming a 24-hour format
    hours = (hours + 1) % 24;
}
```

Listing 2: Implementation of the `incrementHours` Function

II. Constructors and Class Instances

So far, the students know how to manipulate the attributes of an object through functions. But, how the object is created

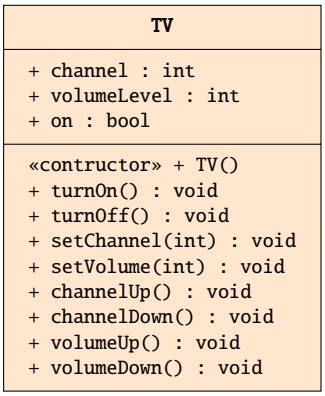


Fig. 2: UML Diagram for a TV with a Constructor

for the first place? Having introduced students to classes, the next natural step for them is the create instances of classes as objects. Consider the UML diagram of a Television (TV) object depicted in Figure 2.

The UML diagram for the TV is conceptually very similar to that of the clock with one exception – the additional **public** function of the namesake. The optional keyword «constructor» is used to emphasize this but not essential for students' understanding (thus, it can be omitted if needed). This new function-like addition (called as the constructor) lets create an object. It is worth emphasizing that the constructors do not have a return type. The implementation of the constructor (definition) is also very similar to the function definitions which students have already learned, with the exception that there is no return type. It is, however, important to emphasize the constructor's utility, which is to initialize (set default values) of new objects upon creation.

As for our example, when a new TV is constructed (or bought from a store), by default, it's in default off state. Furthermore, all new TVs have an initial volume level and an initial channel set by default. In fact, all new TVs have these same exact initial preset values. The constructor, thus, help us set these default values without having to explicitly call a function.

Listing 3 is the UML to C++ to translation for the TV. Given our emphasis is on constructors, here the constructor is implemented within the scope of the class implementation for greater clarity, but can be implemented outside the class scope just like any other class function; important point to focus here is on how to set the default values of an object using a constructor, rather than where the constructor is implemented; For brevity, not all function implementations are listed here except for `setChannel(int)` and `setVolume(int)` functions.

The use and the behavior of the constructor can be easily explained to students using a simple driver program. For example, consider the `main()` function depicted in Listing 3, which creates two TV objections `tv1` and `tv2`. While `tv2` remains untouched, the `channel` and `volumeLevel` of `tv1` are

```cpp
#include <iostream>
using namespace std;

class TV
{
  public:
    int channel;
    int volumeLevel;
    bool on;

    TV()
    {
      channel = 1 //default channel
      volumeLevel = 1 //default volume level
      on = false // default state
    }
    void turnOn();
    void turnOff();
    void setChannel(int);
    void setVolume(int);
    void channelUp();
    void channelDown();
};  .
    .
    .
void TV::setChannel(int newCH)
{
  if(on && newCH >= 1 && newCH < 120)
    channel = newCH;
}
void TV::setVolume(int newVOL)
{
  if(on && newVOL >= 1 && newCH < 100)
    volumeLevel = newVOL;
}   .
    .
    .
    .
int main()
{
  TV tv1; // instance of a class
  TV tv2; // instance of a class

  tv1.setChannel(45);
  tv1.setVolume(6);

  cout << "TV1 channel = " << tv1.channel << endl;
  cout << "TV2 channel = " << tv2.channel << endl;
  return 0;
}
```

Listing 3: C++ Translation of the TV UML Diagram including a `main()` function

changed using the `setChannel(int)` and `setVolume(int)` functions. When their channel values are printed using a couple of simple `cout` print statements, students are able to observe that `tv2` is still on its initial channel value, as shown below.

```
TV1 channel = 45
TV2 channel = 1
```

III. Default Constructors and Overloading

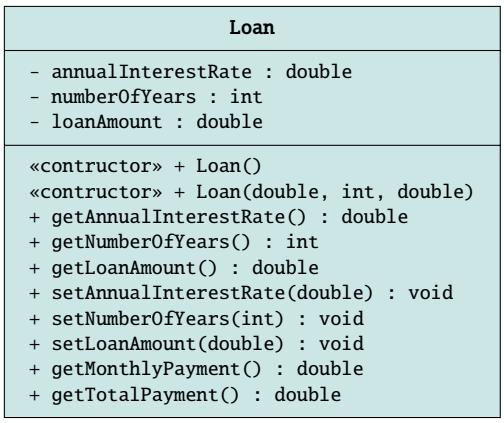


Fig. 3: UML Diagram for a Loan object with Overloaded Constructors

With the TV constructor depicted in Figure 2 and its implementation listed in Listing 3, we see that every TV has the same initial attribute values. In some situations, it is required to set different initial attribute values to different objects. Consider the UML diagram for a Loan class depicted in Figure 3. It has two constructors: one with no parameters and another with parameters. The former one called the *default constructor* is suitable to set attribute values common to majority of loans. The later constructor, which is an *overloaded* constructor, is useful when the attributes are different from the common attributes.

Listing 4 depicts the UML to C++ translation for the Loan object depicted in Figure 3. Just as in Listing 3, function implementations have been skipped for brevity. The important point to emphasize to students during instruction is how the default constructor is pre-populated with default values (just as in the TV() constructor in Listing 3), and the overloaded constructor is distinguished from the default constructor (or any other constructor as for that matter) using a unique permutation of input parameters. Further reinforcement can be provided through testing, such as what's done here in the `main()` function. Two Loan objects are created, one using the default constructor called the `originalLoan`, and another using the overloaded constructor called the `refinanceLoan`. When their private attribute `interestRate` is printed, the object created with the default constructor prints the default value, while the object created with the overloaded constructor prints the user provided value. A sample output of this code execution is provided as follows:

```
Original Rate = 4.25
Refinance Rate = 2.99
```

In general, there could be as many constructors as one may desired, as long as no two constructors share the same function signature (i.e., the same list of parameters). It is worth reminding the students about function overloading rules here, for instance, the name of the parameters does not make the list different, but the type of the parameters does.

```cpp
#include <iostream>
using namespace std;

class loan
{
  private:
    double interestRate;
    int numberOfYears;
    double loanAmount;

  public:
    loan()
    {
      interestRate = 4.25;
      numberOfYears = 30;
      loanAmount = 250,000;
    }
    loan(double rate, int years, double amount)
    {
      interestRate = rate;
      numberOfYears = years;
      loanAmount = amount;
    }
    double getInterestRate();
    int getNumberOfYears();
    double getLoanAmount();
    void setInterestRate(double);
    void setNumberOfYears(int);
    void setLoanAmount(double);
    double getMonthlyPayment();
    double getTotalPayment();
};  .
    .
    .
int main()
{
  // instance of a class with default constructor
  loan originalLoan;
  // instance of a class with overloaded constructor
  loan refinanceLoan(2.99,15,189000);

  cout << "Original Rate = "
       << originalLoan.getInterestRate() << endl;
  cout << "Refinance Rate = "
       << refinanceLoan.getInterestRate() << endl;
  return 0;
}
```

Listing 4: C++ Translation of the Loan UML Diagram with Overloaded Constructors, including a `main()` Function

IV. Inheritance

Now, let's consider a program that maintains an inventory of motor vehicles ranging from motorcycles to trucks for a rental service provider. For each different vehicle, we have set of attributes of which some are common and some other are unique. Similar observations can be made on functions as well where some capabilities are common to all vehicles. To illustrate this point, consider Figure 4 that depicts individual UML diagrams for a motorcycle, a car, and a truck. All three vehicles have two common attributes: some number of

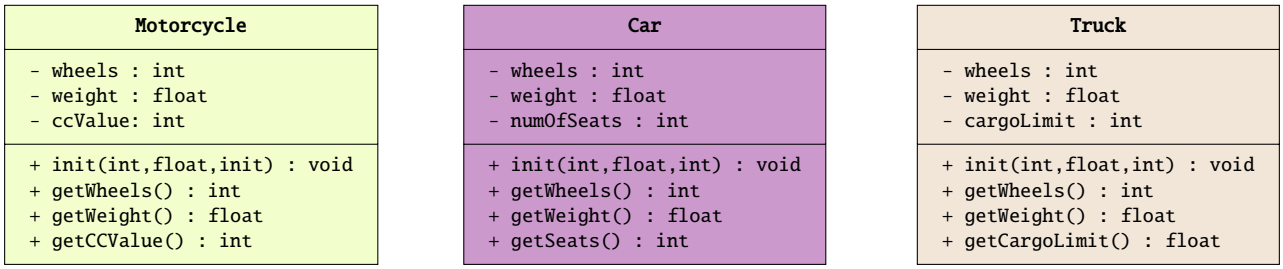


Fig. 4: UML Diagrams of Motorcycle, Car, and Truck Objects

`wheels`, and a `weight`. similarly, all three vehicles also have three common functions: `void init(int,float,int)`[2], `int getWheels(void)`, and `float getWeight(void)`. Additionally, each UML diagram also depicts an attribute unique to each vehicle along with a function to retrieve that attribute value.

An example, such as this one, is an effective way to introduce the concept of Inheritance. By pointing out that redundancies that exist in different vehicles, students can be motivated to understand how a hierarchy is a better way to describe the relationship between them. It can be also used to show how the overall cost of maintaining different vehicle objects can be reduced by reducing redundancies. This natural observation can be then explained using a hierarchy similar to the one depicted in Figure 5. Here, the common attributes and functions from the three vehicles have been removed and aggregated into a new parent object called the `Vehicle`.

At this point, students naturally start questioning the type of access modifier that should be used for the common attributes in the parent object. Since they were private attributes in individual child objects, it makes sense to mark them as private in the parent object. However, in doing so, the child objects lose access to the attributes. This becomes the opportunity to introduce *protected* (#) access modifier.

When common attributes are rearranged using hierarchies, the access modifier of the common attributes are also modified correspondingly; the private access modifier (-) has now turned into a protected access modifier (#). Member functions in derived classes have access to all protected members (e.g., `weight` and `wheels`) and public members (e.g., `void init(int,float,int)`) but, not to private members. Thus, with hierarchical relationships, childs (e.g. Motorcycle, Car, and Truck) "*inherit*" features of the parents (e.g. Vehicle). Effectively, a Car **`is-a`** Vehicle. Similarly, a Motorcycle or a Truck is also **`is-a`** Vehicle. But a Motorcycle is not a Truck or a Car.

The **`is-a`** relationship, which is indicative of Inheritance, depends on where each object lie in the hierarchy; objects in different levels have a **`is-a`** relationship, while objects at the same level do not have a is-a relationship.

[2]An initialization function, similar in concept to a default constructor

V. Related work

It is a common understanding that learning programming is challenging. The reasons for these challenges include unsuitable learning styles, lack of motivation, and need for multiple skills and knowledge [3]. It becomes even more challenging when it comes to object-oriented programming [4]. The instructors experiment with different strategies to make learning exciting and fun, and more importantly improve the retention. This section addresses different strategies that have been experimented to achieve the aforementioned objectives.

One such strategy that teachers have adopted is introducing active learning approaches [5], including teaching in computer labs [6], class time assignment[7], having mandatory lab class, flipped classroom [5], [8], [9], group assignments [10]–[12], 2-stage assignment submissions [11]. Some other experiments consider using other tools such as visual aided programming languages (e.g., Alice and Scratch [13]), game-based approaches [14], different applications (e.g., Mind mapping software [15]) and kids-favorite toys such as Lego®[16]. In addition to the above strategies, some researches introduce different modeling approaches [17] to teach programming, especially object-oriented concepts.

Active learning approaches would be effective in learning as, in most of those cases, the students get a chance to apply their critical thinking. It, however, still needs instructor's supervision and guidance for struggling students which makes it less practical for large class sizes. Alice and Scratch, on the other hand, would work fine in large classes when each student has access to the software, which is feasible. Alice and Scratch would be good platforms to introduce programming as well. Especially, for high-school and middle-school students. They provide a friendly and easy programming environment without the chaos in adhering punctuation and syntax. However, when it comes to college-level programming, students should be able to develop the skills to write programs on their own where they will be using proper punctuation and syntax. At this level, Alice and Scratch do not help that much.

We introduce a novel visual aid into the scope of object-oriented teaching by which students will be able to grasp object-oriented concepts quickly; it is UML diagrams. None of the previous work has been found in this direction. One article

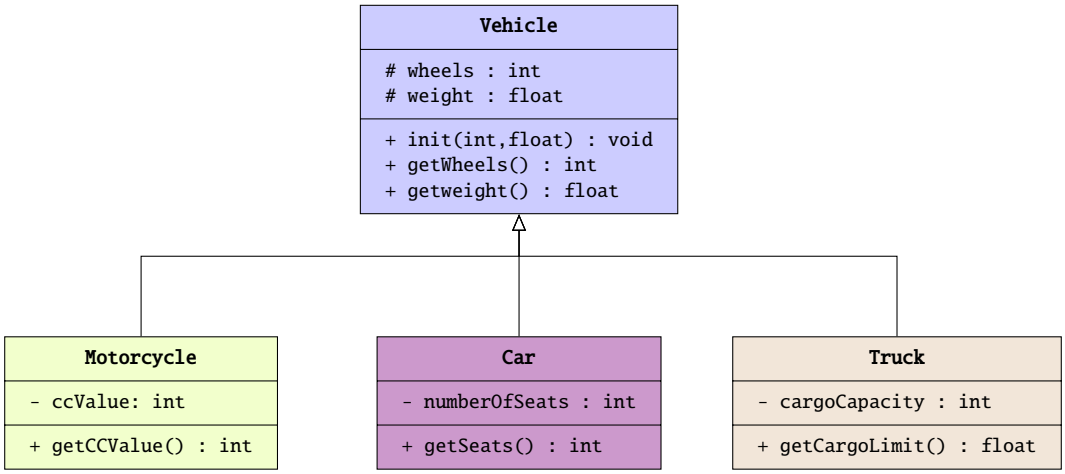


Fig. 5: UML Diagram of Vehicles with Inheritance

[18], however, talks about the experience of teaching OOP and UML to different audiences and talks about them as separate modeling concepts. The article [18] extends their discussion on the suitability of each modeling concepts to different audiences. Our approach, on the other hand, demonstrate how UML's visual representation aspect can be blended in teaching OOP; in fact in C++. This method has several advantages, including, it is scalable for any size of class, it does not require any additional software or accessories that are carried to and from the class. Additionally, showing the concepts all at once (for example, the entire class definition and inheritance), clears away their burden on connecting and relating different pieces of concepts together.

VI. Conclusion

This paper shares experience in using UML diagrams as a way to introduce OOP concepts to first-time course takers. The paper presented several examples of UML diagrams used to introduce classes, constructors, class instances (objects), function overloading, and inheritance. Also presented are an effective way to convert a UML diagram to the corresponding programming code.

References

[1] T. Gaddis, J. Walters, and G. Muganda, *Starting Out with C++: Early Objects*, 7th. USA: Addison-Wesley Publishing Company, 2010, ch. 8.6, ISBN: 0136077749.

[2] W. Savitch, *Absolute C++*, 5th. USA: Addison-Wesley Publishing Company, 2012, ch. 5.4, ISBN: 9780132830713.

[3] T. Jenkins, "On the Difficulty of Learning to Program," in *Loughborough University*, 2002.

[4] J. Bennedsen, M. E. Caspersen, and M. Klling, *Reflections on the Teaching of Programming: Methods and Implementations*, 1st ed. Springer Publishing Company, Incorporated, 2008, ISBN: 3540779337.

[5] D. Chakravorty, M. Pennings, H. Liu, Z. Wei, D. Rodriguez, L. Jordan, D. McMullen, N. Ghaffari, S. Le, D. Rodriquez, C. Buchanan, and N. Gober, "Evaluating active learning approaches for teaching intermediate programming at an early undergraduate level," *The Journal of Computational Science Education*, vol. 10, pp. 61–66, Jan. 2019. DOI: 10.22369/issn.2153-4136/10/1/10.

[6] M. O. Hegazi and M. Alhawarat, "The challenges and the opportunities of teaching the introductory computer programming course: Case study," Oct. 2015. DOI: 10.1109/ECONF.2015.61.

[7] S. Zhuang, H. Wang, W. Zhao, T. Fan, and Y. Zhang, "Practical guidance method for c/c++ teaching reform," Jul. 2015, pp. 834–837. DOI: 10.1109/ICCSE.2015.7250361.

[8] Y. Shi, Y. Ma, J. MacLeod, and H. H. Yang, "College Students' Cognitive Learning Outcomes in Flipped Classroom Instruction: A Meta-Analysis of the Empirical Literature," *Journal of Computers in Education*, May 2019. DOI: 10.1007/s40692-019-00142-8.

[9] G. Akcayir and M. Akçayır, "The Flipped Classroom: A Review of Its Advantages and Challenges," *Computers and Education*, vol. 126, Aug. 2018. DOI: 10.1016/j.compedu.2018.07.021.

[10] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik, "Improving the cs1 experience with pair programming," vol. 35, Jan. 2003, pp. 359–362. DOI: 10.1145/792548.612006.

[11] J. Chen, Y. Cao, L. Du, Y. Ouyang, and L. Shen, "Improve student performance using moderated two-stage projects," Apr. 2019, pp. 201–207. DOI: 10.1145/3300115.3309524.

[12] H. Yuan and Y. Cao, "Hybrid pair programming - a promising alternative to standard pair programming,"

Feb. 2019, pp. 1046–1052. DOI: 10.1145/3287324.3287352.

[13] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, p. 16, Nov. 2010. DOI: 10.1145/1868358.1868363.

[14] Y. S. Wong and M. Yatim, "Computer game as learning and teaching tool for object oriented programming in higher education institution," *Procedia - Social and Behavioral Sciences*, vol. 123, Mar. 2014. DOI: 10.1016/j.sbspro.2014.01.1417.

[15] Y. Liu, Y. Tong, and Y. Yang, "The application of mind mapping into college computer programming teaching," *Procedia Computer Science*, vol. 129, pp. 66–70, Jan. 2018. DOI: 10.1016/j.procs.2018.03.047.

[16] C. Hood and D. Hood, "Teaching programming and language concepts using legos®," vol. 37, Sep. 2005, pp. 19–23. DOI: 10.1145/1151954.1067454.

[17] M. Pedroni and B. Meyer, "Object-Oriented Modeling of Object-Oriented Concepts," in *Teaching Fundamentals Concepts of Informatics*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 155–169, ISBN: 978-3-642-11376-5.

[18] S. Moisan and J.-P. Rigault, "Teaching object-oriented modeling and uml to various audiences," in *Models in Software Engineering*, S. Ghosh, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 40–54, ISBN: 978-3-642-12261-3.