

Intuitiveness of Class and Object Encapsulation

Janina Voigt, Warwick Irwin, Neville Churcher

Abstract--Encapsulation is one of the most fundamental programming language mechanisms available to software developers for managing the complexity of software systems. One might therefore expect clear guidelines and consistent practices to be used in mature programming languages, and particularly in object oriented (OO) languages, with their rich support for encapsulation. However, the encapsulation practices employed by OO developers are surprisingly variable, even within a given OO language. Published advice on how best to use encapsulation is conflicting and little research has been done to determine what developers do in practice and why.

In this work, we focus on one aspect of encapsulation: the encapsulation boundary in OO systems. In the archetypal OO language Smalltalk, object data is private to an object. On the other hand, in statically typed OO languages such as C# and Java, object data is private to a class. This difference has broad implications for software design and maintenance, especially when inheritance is considered. Using a survey of both novice and experienced software developers, we show that the encapsulation boundary supported by mainstream statically typed languages does not coincide with the intuition of most developers.

Index Terms--Encapsulation, Encapsulation Boundary, OO design, Information Hiding

I. INTRODUCTION

Software systems are often large and very complex, making them difficult to comprehend and maintain. Programs commonly contain thousands or even millions of lines of code; far too many for any one person to understand. The difficulties that arise from the sheer size of software are compounded by the complexity that results from coupling between software components. "Programming is about managing complexity", according to Bruce Eckel [3, page6]. Complexity in software systems leads to systems not meeting their specifications, suffering from quality problems, or even outright project failure.

A cardinal strategy used by software designers to control complexity is to decompose systems into loosely coupled components [20]. Parnas formulated this idea as the principle of Information Hiding [15], which encourages designers to hide implementation details so that the rest of the program cannot

depend on them. This reduces the cognitive load on developers because they can ignore hidden details when considering the services offered by a software component, and makes systems more amenable to change because hidden features can be modified without directly affecting client code.

Encapsulation is a programming language mechanism that enables Information Hiding, and so is arguably the most fundamental tool programmers have for managing complexity. Encapsulation mechanisms provide a way of establishing a boundary around a logical module of a system, and of hiding data and implementation details within that boundary to ensure that only the module that owns the information can access and modify it [1], [17].

OO systems provide several levels of encapsulation, usually in addition to conventional source code modules: Packages encapsulate classes, classes and objects encapsulate data and methods, and methods encapsulate algorithms. In this work we are interested in object and class level encapsulation of data, particularly in the presence of inheritance.

II. BACKGROUND

Many design heuristics, principles and guidelines have been proposed to help OO designers, including 61 "golden rules" of OO design introduced by Riel [16], heuristics by John Lakos [12], code smells by Fowler and Beck [4], and the advice of Ralph Johnson and Brian Foote [10]. Since encapsulation is such an important aspect of software design, many of these rules provide explicit or implicit guidance on how to practice encapsulation. Examples include the Separation of Concerns principle [2], the Law of Demeter [13], and several of Riel's heuristics, such as "All data should be hidden within its class" [16]. Advice in this area sometimes conflicts, resulting in confusion on the part of designers, inconsistent code, and ultimately software that is harder to understand and maintain.

While there is universal acceptance of the value of encapsulating data to protect it from the rest of the system, there is no consensus among OO designers on where the encapsulation boundary should lie. Encapsulation is enforced in two main ways in modern OO programming languages; we will refer to them as *object encapsulation* and *class encapsulation*.

Object encapsulation is commonly used by dynamically typed languages, including Smalltalk [11] and Ruby [19]. In these languages, data is private to an object. This means that when an object contains data, only that object has the right to access and modify this data; it cannot be directly changed by any other object, regardless of that object's class.

In contrast, today's dominant statically typed OO programming languages, including Java [5], C# [6] and

Janina Voigt is a postgraduate student at the University of Canterbury, Christchurch, New Zealand (e-mail: jvo24@ student.canterbury.ac.nz).

Warwick Irwin and Neville Churcher are both with the University of Canterbury, Christchurch, NZ (e-mail: warwick.irwin@canterbury.ac.nz, neville.churcher@canterbury.ac.nz).

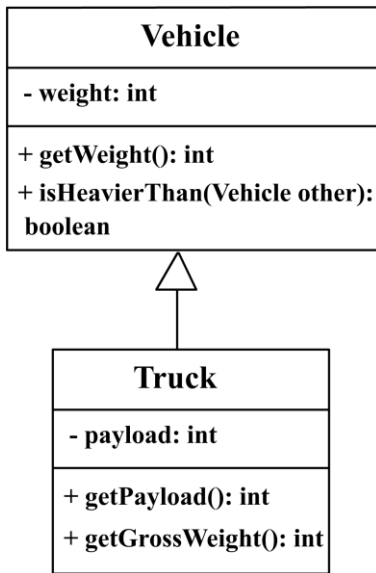


Fig. 1. A class view

C++[18], enforce class encapsulation: the most tightly protected data is private not to an object but to a class. By placing the encapsulation boundary around classes, these languages allow objects of the same class to access each others' private data.

This difference becomes more important when inheritance is considered. When using object encapsulation, a single object can access all of that object's data, regardless of which class it was inherited from. When using class encapsulation, objects cannot access private members inherited from a superclass, even though they are part of the same object. The class encapsulation boundary cuts the object, making the inherited part inaccessible to the derived part.

The protected access mechanism provided by many current OO programming languages allows an object of a subclass to access protected parts inherited from a superclass. This enables an approximation of object encapsulation, because if all inherited members are protected an object is able to access all of its contents. Nevertheless, this is still a variation of class encapsulation rather than true object encapsulation, because other objects of the same class can also access the object's contents. In some programming languages such as Java, protected access also confers access rights on all objects of other classes in the same package.

The two different types of encapsulation represent very different design philosophies. They are best explained using an example. Vehicles have a weight and can compare their weight to that of another Vehicle using the method `isHeavierThan(Vehicle other)`. The class also defines a simple accessor method (or getter) for the weight field. Trucks inherit from Vehicles and add their own field `payload`, which describes the maximum load a truck is allowed to carry. In addition to this field, the Truck class also defines a getter method for the `payload` field and a method called `getGrossWeight()` that can calculate the total

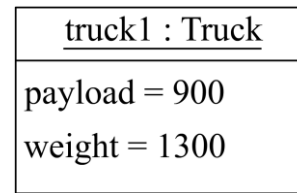


Fig. 2. An object view

weight of a Truck: its net weight plus its payload.

Fig. 1 shows a class encapsulation view of the example. Vehicle and Truck are different classes, each with their own sets of attributes and methods. These classes are the building blocks of the program when it is constructed. Class encapsulation reflects a designer's mindset oriented around static, compile-time concepts. According to this mindset, it makes little sense to allow classes to access other classes' private members.

Fig. 2 shows the object encapsulation view. In this paradigm, the Truck object is a single entity, in part defined by the Truck class and in part by Vehicle. This mindset is oriented around the runtime concept of objects. For this way of thinking, it does not make sense for a Truck to be able to access only a part of itself.

Both types of encapsulation have advantages and disadvantages. We have observed that many students who use Java are surprised when they find out that objects of the same class can access each others' private data. Many students seem to find object encapsulation intuitively correct, and assume that programming languages support it.

However, while object encapsulation may be more intuitive, Snyder (and others) argue that it removes all advantages to be gained from the use of encapsulation [17]. He suggests that by allowing access to data in another class—which may have been written by a different developer—the maintainability of the system is compromised. The reason he gives is that, should the other developer wish to change the internal data representation for that class, the subclass accessing that data will also be affected.

Similar arguments can be made in defence of object encapsulation, however. The extensibility and reusability of a software system might be enhanced by allowing objects of subclasses to access inherited implementation details; this gives them freedom to override or reuse existing members in ways that would be prevented by class encapsulation. Again, this is effectively a philosophical choice: object encapsulation is a permissive approach intended to maximise subclass freedoms, while class encapsulation favours tighter control. An issue that may influence a designer's choice here is the question of whether inheritance is innately such a strong dependency that it does not make sense to try to isolate a subclass from changes to its superclass.

As a first step in investigating whether our speculations on the encapsulation boundary are valid, we decided to conduct a survey to clarify how programmers use encapsulation in the

simple `Vehicle` scenario described above.

III. SURVEY

We designed a survey to investigate how novice and experienced software developers practise encapsulation. This section describes the goals we were trying to achieve, the survey participants and the tasks in the survey.

A. Goals

From personal experience and from working closely with computer science students, we suspect that many students learning object oriented programming using a programming language like Java or C++ tend to assume that `private` data is private to an object and are surprised and in some cases shocked when they learn that it is instead private to a class. Many of them feel uncomfortable when accessing the `private` data in another object of the same class. It seems that this conflicts with their intuitive expectation of where the encapsulation boundary should lie.

However, over time many appear to adapt to some degree to the tools a programming language provides them. They start to access `private` data from another object of the same class on occasion, particularly in places like the `equals()` method in Java, despite the fact that it conflicts with their intuition. We have heard people justify these decisions by saying that accessing `private` data is more efficient and quicker to code.

We decided to conduct a formal survey involving a number of novice and experienced programmers to clarify their encapsulation practices. We surveyed 34 undergraduate students, 9 postgraduate students and 12 professional developers about their practices of encapsulation. We expected to show that, while professional developers have adapted to the class encapsulation mechanism provided by most modern programming languages, object encapsulation makes intuitively more sense to novice programmers.

B. Participants

The survey was conducted with both undergraduate and postgraduate university students and professional software developers. The undergraduates were volunteers from two computer science courses at the University of Canterbury. The first course was a second-year course about computational theory. The students in the course had just completed their first year of computer science, including an introduction to Java but had relatively little experience, having not yet completed a programming project other than the usual small CS1 and CS2 assignments. The second class we surveyed was a third-year software engineering course. These students had all completed a second-year software engineering course which included a group project in Java where they developed software for a real client over a period of 6 weeks. We also surveyed postgraduate students who all had a substantial amount of experience using Java.

In addition to surveying students, we surveyed 12 professional software developers who routinely used C# as part of their work. They were likely to be far more proficient

programmers than undergraduate students and more aware of OO design principles, having programmed professionally from anywhere between 2 and 20 years.

C. Task

We carefully designed the survey to allow us to infer the encapsulation practices and principles of participants rather than asking them directly. We did not want participants to over-think their replies but rather to act as they would when programming. The two main parts of the survey can be seen in Fig. 3. The corresponding class diagram is the one already presented in Fig. 1.

The survey consisted of two main questions that were designed to exemplify the difference between object and class encapsulation. For each question, we presented a small class containing a few fields and methods. We then asked developers to complete a new method by choosing between three alternatives. Each alternative completed the method in a way that achieved similar functionality. However, the difference between the options was that some used getters to access fields while others accessed data directly.

For each of the two questions, we asked developers to rank the three options from 'best' to 'worst' and to explain the ranking they decided on.

The first question focused on the question of whether an object should be able to access the `private` data of another object of the same class; this is allowed in class encapsulation but not in object encapsulation. Subjects ranked, in order of preference, three given options for the completion of the `isHeavierThan(Vehicle other)` method which compared the weight of one `Vehicle` object to that of another `Vehicle`.

The second question focused on the question of whether an object should be able to access inherited `private` data; this is allowed in object encapsulation but not in class encapsulation.

Subjects ranked three given options to complete a method called `getGrossWeight()`, which returned the sum of the truck's weight (inherited from `Vehicle`) and payload (locally defined in `Truck`). Again, the only difference between the options was that some used getters while others accessed variables directly.

Many programmers will automatically invoke getters, if they exist, rather than accessing fields directly, and this convention might overpower any preference for a particular encapsulation boundary. Similarly, if programmers automatically access fields directly whenever possible, this convention may dominate any single encapsulation boundary. Both questions asked subjects to rank the alternatives, rather than simply pick a favourite, so that in a number of cases we were able to determine their encapsulation boundary preferences, even if they always favoured getters or direct access. In other cases, their comments gave clues about their way of thinking.

In addition to the two main questions, we included two very simple coding exercises—asking participants to write a `toString()` method for the `Vehicle` and `Truck` classes—to test the competence of the participants. These

Consider the following class Vehicle:

```
public class Vehicle {
    private int weight;

    public int getWeight() {
        return weight;
    }

    public boolean isHeavierThan(Vehicle other) {
        //Something goes here
    }
}
```

Now we want to complete the code for the `isHeavierThan(Vehicle other)` method of the Vehicle class.

Here are several ways in which we could complete this method:

Option 1

```
public boolean isHeavierThan(Vehicle other) {
    return other.weight < this.weight;
}
```

Option 2

```
public boolean isHeavierThan(Vehicle other) {
    return other.getWeight() < this.weight;
}
```

Option 3

```
public boolean isHeavierThan(Vehicle other) {
    return other.getWeight() < this.getWeight();
}
```

We will now extend our design by adding a second class, `Truck`, which is a subclass of `Vehicle`. The `Truck` class contains a field `payload` storing the maximum load the truck is allowed to carry.

```
public class Truck extends Vehicle {
    private int payload;

    public int getPayload() {
        return payload;
    }

    public int getGrossWeight() {
        //Something goes here
    }
}
```

Now we want to complete the code in the `getGrossWeight()` method of the `Truck` class. This method is supposed to calculate the gross weight of the truck, i.e. the weight of the truck plus the maximum load it can carry. Here are several different ways in which this method could be written:

Option 1

```
public int getGrossWeight() {
    return weight + payload;
}
```

Option 2

```
public int getGrossWeight() {
    return getWeight() + payload;
}
```

Option 3

```
public int getGrossWeight() {
    return getWeight() + getPayload();
}
```

Fig. 3. Main parts of the Encapsulation Questionnaire

questions enabled us to eliminate participants who did not have enough basic programming knowledge to competently complete the questionnaire. On the basis of these two questions, we eliminated two of the responses.

For the professional software developers, we also included a question about their previous programming experience, including their first programming language and the amount of time they had used C# or VB.NET. We translated the questionnaire from Java to C#, making sure that it was semantically identical to the Java questionnaire.

IV. RESULTS

We classified respondents by the encapsulation practices they preferred, as shown in Fig. 4. The results from students provide evidence to support our theory that novice programmers tend to find object encapsulation much more intuitive than class encapsulation. The undergraduate students could be divided into four major groups:

- Those who preferred using getters rather than accessing data directly. (No single encapsulation

boundary.)

- Those who practised object encapsulation.
- Those who preferred accessing data directly rather than using getters. (No single encapsulation boundary.)
- Those who did not mind whether getters were used or data was accessed directly as long as the approach used was consistent. (No single encapsulation boundary.)

More than half of the students (58 %) preferred using getters to accessing data directly. This is not surprising since they have been taught in class that getters make a system more maintainable. They commented that using getters was better style, safer and made the system more maintainable, and also said that getter methods encapsulate private data.

The second largest group (24%), practised object encapsulation. They preferred to access private data in a superclass directly even though this would cause a compile error in Java, but did not want to access private data from another object of the same class. From their comments, it was evident that they truly believed that this was what Java allowed. They

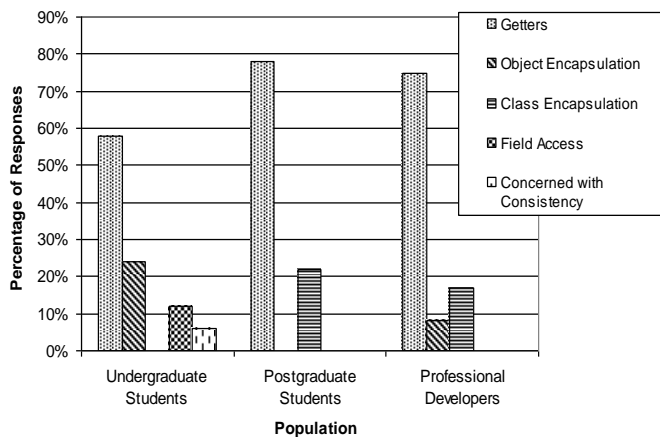


Fig. 4. Classification of survey responses

often commented that accessing private data in another object of the same class was not possible.

The remaining groups were both small, with about 12 percent of students preferring to access data directly rather than using getters. They usually commented that this was more efficient. The last group of students (6%) were simply concerned with keeping the coding approach as consistent as possible.

Notably, there were no students who practised pure class encapsulation; that is, no one accessed the private data in another object of the same class but not the private data in a superclass.

Many of the students used getters to access data and therefore used neither object nor class encapsulation explicitly. However, in a number of cases their rankings still showed which they would choose if getters were unavailable, and their comments provided clarification of their reasoning. Using this information, we could classify respondents by whether they showed object or class encapsulation tendencies or both. The results of this additional analysis can be seen in Fig. 5.

For undergraduate students, two responses clearly showed partial object encapsulation thinking, with students commenting (incorrectly) that it was not possible to access a private field of another object of the same class. Another three responses showed a partial tendency toward class encapsulation, showing that these students were aware of Java's approach to encapsulation; they commented that a private field in a superclass could not be accessed directly. This is not a surprising response since they have been taught this in class. The remaining two students occupied an uneasy middle ground, showing tendencies towards both types of encapsulation, and were clearly confused about what Java allows.

We wanted to compare the way novice programmers think to how more experienced and professional software engineers think about encapsulation. We surveyed nine postgraduate students all of whom were very proficient in Java, and twelve professional software developers who were experienced .NET developers.

Interestingly, we found that none of the postgraduate students were using object encapsulation, but two used class encapsulation (22%). They commented that accessing the fields

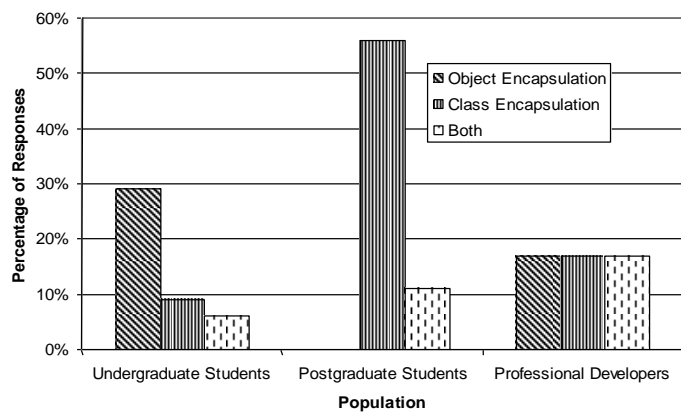


Fig. 5. Encapsulation tendencies for the three populations

of another object of the same class directly was simple and valid while accessing the private fields in a superclass was not allowed. This clearly shows that they think differently from novice programmers. The remaining postgraduates (78%) preferred always using getters to support encapsulation.

We again had a closer look at the responses of the postgraduate students who used getters to see if we could infer more about their way of thinking. Three of the seven respondents who used getters showed definite class encapsulation tendencies, while another one showed tendencies both ways, and appeared to be confused about encapsulation in Java.

We saw a similar effect when we surveyed twelve professional .NET developers. The largest group (75%) again liked to always use getters. Two respondents (17%) used pure class encapsulation, clearly demonstrating that they were aware of what was valid in C#. Both commented that accessing private fields in a superclass was not valid and would not compile. One developer with 5 years experience using C# still believed that object encapsulation was correct.

A closer look at the responses of developers who used getters showed that even some professional developers are not completely comfortable with encapsulation in C#. One developer showed object encapsulation tendencies stating that accessing private fields of another object of the same class would cause a compile-time error. Two more developers showed both object and class encapsulation tendencies in their survey and appeared generally unsure about what was allowed and what was not.

V. DISCUSSION

The results from the student survey clearly support our contention that novice programmers find object encapsulation more intuitive than class encapsulation. More than a quarter of the undergraduate students we surveyed with as much as two years of programming experience still believed that Java effectively supports object encapsulation. In addition, no students were comfortable using what Java provides: pure class encapsulation. Some students showed signs of being aware of

encapsulation mechanisms in Java but no one wanted to use them.

This result has important implications because it shows that novice programmers are uncomfortable with the encapsulation mechanisms provided by many modern programming languages, including Java and C#. Object encapsulation, not class encapsulation, appears to make sense to them.

Even some postgraduate students and professional software engineers, all of whom were proficient in either Java or C#, showed signs of unease and confusion about the encapsulation mechanisms provided. Some did not appear to be entirely sure about what was allowed and what was not despite years of programming experience and the very basic nature of the exercises.

However, there was a clear sign that a number of them had adapted to what the programming language they were using provided them with, because around 20 percent used class encapsulation.

Overall, we believe that our survey shows that class encapsulation as provided by many modern programming languages is not what novice programmers expect and can confuse even experienced developers.

VI. CONCLUSION AND FUTURE WORK

We have conducted a survey amongst both novice and experienced programmers to determine how they practice encapsulation. We were particularly interested to find out if they preferred object or class encapsulation.

The difference between object and class encapsulation is that the encapsulation boundary is in a different place, making a different set of operations legal and illegal. Most modern programming languages like Java and C# use class encapsulation, while some languages like Ruby, Smalltalk and Java Script use object encapsulation.

Overall, our survey found that the class encapsulation mechanism provided by most of today's mainstream programming languages is unintuitive for novice programmers. While over time programmers appear to adapt to what the language allows them to do, there is still confusion amongst some experienced programmers as to what is allowed and what is not. We therefore argue that class encapsulation as provided by C# or Java is not what programmers intuitively expect or want.

This work is only the first step in our investigation into encapsulation practices. The results of this survey have provided us with useful insights into the issues surrounding encapsulation practices which will inform the next phase of our research. Because encapsulation is so fundamental, the lack of consistency uncovered by the survey has far-reaching consequences that impact on virtually all other principles and guidelines of OO design. We are currently developing tools to perform a quantitative analysis of encapsulation practices by extracting relevant data from the Qualitas Code Corpus of Java programs from the University of Auckland [14]. We have previously developed a very accurate semantic model for Java

called Java Symbol Table (JST) [7] – [9] which we plan to use in the analysis of these programs. The aim will be to determine whether object or class encapsulation is mostly used in real software and whether direct accesses of data are common. We expect the results to provide useful material for users and designers of OO languages.

REFERENCES

- [1] A. Cohen, "Data abstraction, data encapsulation and object-oriented programming," *ACM SIGPLAN Notices*, vol. 19, pp. 31–35, 1984.
- [2] E. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*, pp. 60–66, 1982.
- [3] B. Eckel and Z. Sysop, *Thinking in Java*. Upper Saddle River, NJ: Prentice Hall, 1998.
- [4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [5] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.
- [6] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde, *The C# Programming Language*. Addison-Wesley Professional, 2008.
- [7] W. Irwin, "Understanding and improving object-oriented software through static software analysis", PhD thesis, University of Canterbury, 2007.
- [8] W. Irwin and N. Churcher, "Object oriented metrics: Precision tools and configurable visualisations," *METRICS2003: 9th IEEE Symposium on Software Metrics*, Press, pp. 112–123, 2003.
- [9] W. Irwin, C. Cook, and N. Churcher, "Parsing and semantic modelling for software engineering applications," *ASWEC '05: Proceedings of the 2005 Australian Conference on Software Engineering*, Washington, DC, USA: IEEE Computer Society, pp. 180–189, 2005.
- [10] R. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22 – 35, 1988.
- [11] A. Kay, "The early history of Smalltalk," in *History of Programming Languages*, New York, NY, USA: ACM, 1996, pp. 511–598.
- [12] J. Lakos, *Large-Scale C++ Software Design*. Addison-Wesley, 1996.
- [13] K. Lieberherr and I. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, pp. 38 – 48, 1989.
- [14] H. Melton and E. Tempero, "The CRSS metric for package design quality," *ACSC '07: Proceedings of the thirtieth Australasian conference on Computer science*, Darlinghurst, Australia: Australian Computer Society, Inc., pp. 201–210, 2007.
- [15] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, 1972.
- [16] A. Riel, *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.
- [17] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," *OOPSLA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, New York, NY, USA: ACM, pp. 38–45, 1986.
- [18] B. Stroustrup, *The C++ Programming Language, Third Edition*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.
- [19] D. Thomas and A. Hunt, *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley, 2001.
- [20] E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1979.