Teaching Encapsulation and Modularity in Object-Oriented Languages with Access Graphs

Gilles Ardourel, Marianne Huchard

HAL Id: lirmm-00191529

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00191529

Submitted on 26 Nov 2007

Teaching Encapsulation and Modularity in Object-Oriented Languages with Access Graphs

G. Ardourel, M. Huchard

LIRMM

161 rue Ada

34392 Montpellier Cedex 5 France

ardourel@lirmm.fr,huchard@lirmm.fr

April 9, 2002

Abstract

Encapsulation and modularity are supported by various static access control mechanisms that manage implementation hiding and define interfaces adapted to different client profiles. Programming languages use numerous and very different mechanisms, the cumulative application of which is sometimes confusing and hard to predict. Teaching these concepts requires precise and comparable definitions of the mechanisms used by the languages taught. We present here our experiences with access graphs, a notation we use for teaching encapsulation and modularity in object-oriented languages.

1 Introduction

As noticed by [5], "Access control is actually a complex topic that aids in defining well-structured software and therefore in reasoning about correct software". Access control mechanisms play indeed a key role in modular and encapsulated software organizations [6].

Teaching Access control is not easy, and we will discuss here:

- issues in teaching access control,

- access graphs, a notation for access control,

- a teaching experience with access graphs,

- evaluation of access control knowledge and documentation.

1.1 Teaching Issues

Teaching how to use static access control to students is not the trivial task that one may think, and there are several aspects that hinder their understanding and use:

- Languages are evolving fast and documentations are often informal enough to admit several interpretations[1].
- Mechanism intrication can lead to access rights really difficult to understand and predict.
- Modularity and inheritance have conflicting rules concerning access control, as observed by [9].
- Case tools and design methodologies generally propose a rough schema of access control modeling essentially based on the public/private separation.
- There is no general language, independent from the syntax of programming languages, allowing to model and express any access control situation.

---

[1]See Java Spec Report http://www.ergnosis.com/java-spec-report, Sections 15.11.12, 8.2, or 6.6.2.1 for ambiguities in Java access control description.

The last two points are the more damageable to the students, who read *public*, *private* or *protected* in manuals and in UML[4] documentation, without seeing any precise definition of these access controls mechanisms for the languages they use. Furthermore, languages like Eiffel [7] or Ada [10] use different principles than C++ and Java whose syntax has been borrowed by UML.

The point of view of UML is that the semantics depends on the programming language [8].

This leaves the teacher with the difficult task of expressing precisely and accurately the semantics of each mechanisms in each language. To this end, we propose the Access Graphs notation [2] which is the base of a prototype CASE tool for handling Encapsulation [1].

2 Access Graphs

Our purpose is to help intuition and reasoning on static access by providing an easy-to-read representation, giving access control a visual notation with a precise semantics[2] . The idea is to bring to access control the same readability as the one given by UML to other aspects of design (classes, associations, collaborations, etc.). An access graph is a graph were nodes represent classes or set of classes, and edges represent accesses from one node to another. Edges are labelled by a set of properties. To differentiate between accesses from an instance to itself[3] and others accesses to the instance, we prefix the set of properties with either `I:` or `C:`. An edge can have a condition expressed on its starting and ending points, if necessary.

We use three different kinds of access graphs when explaining access control:

- authorized access graphs for an access control mechanism,

- authorized access graphs for a class hierarchy,

- effective access graphs for a class hierarchy.

The figures 1 2 and 3 respectively represent the authorized access graphs for the *public*, *protected* and *private* mechanisms in Java[4]
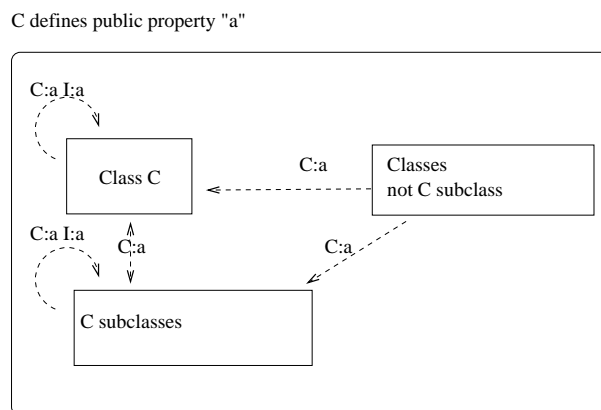

Figure 1: Java public mechanism

3 Teaching with access graphs

Access graphs were used when teaching Java and Eiffel to BSC and MSC students.

They provided several benefits, at the cost of the time needed to explain the notation used and what an access is. The students were already marginally familiar with graph based notation like UML, and having an edge representing an access was not difficult to grasp.

---

[2]Precise semantics is given in [2].

[3]via self or this keywords, for instance

[4]JDK 1.0 and 1.2 had different behaviors for the protected mechanism. Default visibility was not included because it was not relevant to the section 4.
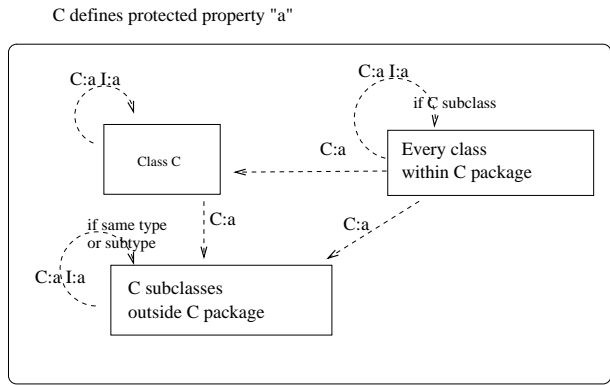
C defines protected property "a"

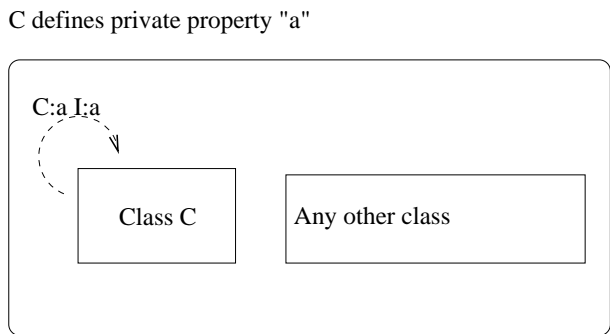Figure 2: Java protected mechanism

C defines private property "a"

Figure 3: Java private mechanism

Access graphs provided a clarified and unambiguous view on access control, leaving less space to erroneous interpretations than the textual documentations used.

Since the access graphs are not language specific, they helped the students to understand access control beyond the simple scope of learning the three keywords shown. We had more feedback from the students that learned with access graphs than from those who had more traditional definitions. The students were able to discuss spontaneously the implication of the definition of each keyword on their future implementations.

Furthermore, those who had previous experience with other OO languages asked for comparisons and identified the differences with Java more precisely.

4   Evaluation of access graphs

The students examinations did not focused on access control, so it was not a good way to know how effective for understanding access control were the access graphs. We wanted to know how they were known (from people that had not used access graphs), and if they were sufficiently expressive to be useful without interaction with a teacher. As an experiment, we did a web poll[5] on static access control.

4.1   the poll

The first step shows the following C++ code, and let the user check the validity of each access.

```
class A {
public:
        int pub;
protected:
        int pro;
private:
```

---

[5]Currently only in french, to be translated within days. ((http::://www.lirmm.fr/ ardourel/Rech/Expe)

```
      int pri;

public :
 virtual void swap(A&)=0;
};

class B : public A {
public :
  virtual void swap (A&);
};

void B::swap(A& a) {
int t1=a.pub;  //line 1
int t2=b.pro;  //line 2
int t3=a.pri;  //line 3

a.pub=pub;  //line 4
a.pro=pro;  //line 5
a.pri=pri;  //line 6

pub=t1;  //line 7
pro=t2;  //line 8
pri=t3;  //line 9
}
```

The intent of this step was figuring out how well were understood access control. It gave us more interesting result than we thought(see results 4.2).

The second part of the test shows a documentation about the `public`, `protected`, and `private` modifiers, in either Text coming from [3] or with the figures 1, 2 and 3. The documentation was randomly selected. Below the documentation is a set of Java classes quite similar[6] to the previous exercise: A class A and a subclass B of A in another package. Another class C in the same package as A is added to take into account this important element of Java. After validation, the user get the same exercise with the other documentation, and can change its previous choices.

4.2   results

There were 61 people loading the poll, 32 answered the first part, and 25 answered all the three parts. People who answered "Unknown language" for a language were not taken into account for the statistics concerning this language[7] We expected two main kinds of error:

- accepting the line 9 (private access)

- accepting lines 2 and 5 (protected access)

But we got another one rather surprising: several users gave us different answers for lines 1-3 than for lines 3-6, that is distinguishing read and write accesses to attributes. We called this one the "read/write" error.

	number	OK	protected	private	read/write
C++	24	4	18	10	5
only C++	8	3	4	3	0

The main information was the lack of knowledge of the protected mechanism the users have.

In the second test, another kind of error could be found in the class C: rejecting accesses to protected properties of A and B (package error). Following are the results of this test.

	number	OK	protected	private	read/write	package
Java (total)	25	4	17	14	3	16
Java (Text)	14	0	11	9	2	9
Java (Graphs)	11	4	6	5	1	7

---

[6]The similarity of code was intended. The accesses allowed are exactly the same for this particular example.
[7]Hence the number 24 for C++ and 25 for Java.

The first line is very disappointing: having[8] a documentation doesn't seem to reduce the number of errors made. However, the comparison of the results between the two documentations seems to indicate that Graph documentation works better.

By focusing our attention only on the users that changed their answers between the two documentation, we had an even more positive result concerning the graph documentation.

- They were five users that increased their score: four were using Text before Access Graphs, and only one was using Access Graphs before Text.

- They were five users that decreased their score: four were using Access Graphs before Text, and only one was using Text before Access Graphs.

Considering that almost all the users were seeing access graphs for the very first time, we find the result satisfactory and we will make larger scale polls to evaluate programming knowledges and documentation techniques.

5   conclusion

Graphic representation as proven itself useful for describing software. We believe that software engineering concepts can also benefit from non-textual visualization.

Using an intuitive language independent notation for explaining how a concept works in a specific language helps the student to keep the concept in mind while understanding the mechanism.

As an example, we described here the access graph notation and our experience with it. It helped students understanding the encapsulation concept beyond the mechanisms. Furthermore, access graphs for hierarchies are close enough from collaboration diagrams to make the transition and comparison between design and implementation easy.

We think that language independent CASE tools allowing the manipulation and definition of software engineering concepts should be very useful for teaching programming, as opposed as teaching how to code in a specific language.

References

[1] G. Ardourel and M. Huchard. AGATE, Access Graph bAsed Tools for handling Encapsulation . In *Proceedings of the 16th IEEE International Conference Automated Software Engineering (ASE) 26-29 Nov. 2001 San Diego California*, pages 311–314.

[2] G. Ardourel and M. Huchard. Access graphs: Another view on static access control for a better understanding and use. *Journal of Object Technology*, 2002. Accepted for publication.

[3] K. Arnold and J. Gosling. *The Java Programming Language Second Edition*. Addison Wesley, 1998.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison - Wesley, 1999.

[5] I. Joyner. *Objects Unencapsulated, Java, Eiffel and C++*. Prentice Hall, 1999.

[6] B. Meyer. *Object-oriented Software Construction*. Englewood Cliffs NJ: Prentice Hall, 1988.

[7] B. Meyer. *Eiffel, The Language*. Prentice Hall - Object-Oriented Series, 1992.

[8] Rational Software Corporation. *UML v 1.3 Documentation*. http://www.rational.com/uml/resources/documentation/index.jtmpl.

[9] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'86*, 21(11):38–45, 1986.

[10] S. T. Taft and R. A. Duff. *The Ada 95 Reference Manual*, volume 1246. Lecture Notes in Computer Science, Springer-Verlag, 1997.

---

[8]It's very difficult to know if it was used with such low level of detail.