ECOOP 2007 - Author Response form for Papers

Note 1: Netscape 4.x users: Do not resize your browser's window once you've started to fill out this form! You will lose what you typed.
Note 2: Responses consisting of more than 500 words will NOT be stored! Please enable JavaScript to use a word counter.

Title	Towards More Sophisticated Access Control in C++
authors	Jozsef Mihalicza, Eotvos Lorand University, jmihalicza@gmail.com Norbert Pataki, Eotvos Lorand University, patakino@elte.hu Adam Sipos, Eotvos Lorand University, shp@elte.hu Zoltan Porkolab, Eotvos Lorand University, gsd@elte.hu

First reviewer's review

>>> Summary of the submission <<<

The paper discusses "sophisticated" access control mechanisms not directly supported by C++ (access groups, roles, and inheritance driven) and shows how they can be expressed using template meta-programming. The techniques are for "large-scale projects where fine-tumed access is crucial."

Their solutions require that callers and callees follow some guidelines in writing code to benefit from the refined access control mechanism. The solution imposes no run-time overhead in most scenarios while adding some minimal run-time overhead in a few cases. Great attention is paid to non-intrusiveness of the approach and the support of incremental development.

>>> Evaluation <<<

My main problem with the paper is that it takes the need for "sophisticated access control" for "large projects" for granted. The opposing view (as expressed in most programming language designs) is that for a technique to scale it has to be simple. Except for a brief reference to Eiffel, no argument is made for the need for the added complexity of the access rules ("added complexity" is simply another way of saying "sophisticated" with a different emotional impact).

Given that I consider the need for more sophisticated/complex access rules unproven (and counter to my experience), I would have liked to see a serious argument or at least a few specific reference to serious work on that topic.

Basically the survey in section 1 and 2 is superficial and presupposes the answer (i.e., that sophisticated/complex mechanism are good or necessary). If the paper is accepted, these sections must be rewritten to be less value laden (in statements and vocabulary) or loaded with references to research supporting the value judgements made.

Section 4.1: The criticsms that a friend grants access to a whole class rather to a specified subset of a class is crucial. If it is true, many of the needs addressed follows. The opposing view is that it is meaningless to grant access to a subset of a representation because if you could write separate invariants for the subsets then the class is really an ill-designed conglomerate of roles. This point should be addressed explicitly.

Should the following be read as primarily related to operations (functions) and member types? Or are refinements of access control to data members also considered valuable?

Why can't we just say ''If you want access to a subset of the functions of a class, either define a superclass with a restricted interface or an independent "access class" with forwarding functions?'' Are the mechanisms for automating sophisticated/complex access controls necessary?

From this point on, I will assume that there is a need for the sophisticated/complex mechanisms:

Basically, the way the language is used to express access policies is sound, though I would have liked to see a real argument about why it would scale to projects involving dozens of people. How can it be taught? How are misuses detected and corrected? Is the quality of error reporting adequate for real examples? How does the techniques scale in terms of compile time?

The use of language facilities to emulate access control (at least to this extent) is reasonable novel and reasonably well done. My guess is that the implementation of the ideas could be refined based on user experience (though I'm not willing to guess in which way).

Has the set of mechanisms ever been used by others than its authors? If so, please give examples and some idea of the experience.

The explanation in section 3 of why friendship relation is neither inherited nor transitive is rather incomplete and needs to be improved upon. Also the explanation in section 4.5 of similarity between exception specification and the relationship between enlisted privileged clients and inheritance is not clear and needs to be reviewed.

The discussion of access by role in 4.6 misses out how roles will be granted: it seems that any newly added class can be granted any role it wants and hence get access to any member with role based access control.

Section 5: The discussion of similarities between improper usage of const_cast etc. and improper usage of proposed refined access control mechanism seem to be an interesting one and should not be skipped in one sentence.

Using macros to hide implementation is common, but widely considered undesirable.

Section 5.2 talks about limitation of the approach with the key, however it is not clear from the subsequent text what those limitations are. Please elaborate.

The exposition needs work: it could be much clearer throughout.

Typos:

Page 3: the implementation is based ON template metaprogramming techniques.
Page 3: Since all the visibilITy controls happen...
Page 3: In C++ we can use keyword friend TO GRANT access to private...
Page 4: access control is determined dynamically AT runtime...
Page 5: C# also supports ADDING access modifier to global classes.
Page 8: protected inheritance is similar to private inheritance IN many ways.
Page 8: "the derived classes of the direved class" can be rephrased better.
Page 8: in programming languages like __ C++ IT is possible to define...
Page 8: frequently called as "namespace functions" - not sure about the terminology used here, please provide references to other works, where they are called like this in your context.
Page 9: IN some cases the reason is technical, ...
Page 11: IN some cases the derived class ...
Page 11: Methods in THE base class ... thus not accessible for the derived
Page 15: to implement__ the IsInNamespace predicate for a given namespace.
Page 16: In the example in 6.1 it is not clear that template function restricted is actually a templated member function, because the definition of class C was removed from the source.
Page 19: ... mechanisms separately, but they often haVE to be used together.
Pages 2, 6, 7, 11: use of "an other" should probably be changed to "another"
A lot of missing citations.

Second reviewer's review

>>> Summary of the submission <<<

This paper introduces a more sophisticated access control mechanism for the language C++. It is motivated by the importance of encapsulation to OO programming and the fact that the traditional C++ access control mechanism does not support the fine granularity that is needed for providing strong encapsulation in more advanced examples.

After giving an overview of access strategies in various OO programming languages and illustrating the reqiurements for a fine-grained access control mechanism using several examples, the authors introduce their advanced access mechanism for C++. This mechanism is based on tenmplates and does not require any modifications of the language/compiler. As an evaluation, the authors then revisit the requirements stated previously and show how they are addressed by the proposed solution.

>>> Evaluation <<<

This paper is written in a pretty concrete and direct manner, which makes it easy to read. It contains all the pieces that are needed for a solid paper: It addresses a clear problem that is illustrated using several examples, there is a pretty pragmatic solution that is actually implemented, and finally there is an evaluation of the solution against the requirements derived from the problems.

The main limitation of this paper is the fact that the solution is quite specific as it only applies to C++. Furthermore, the the solution is creative, it doesn’t seem the most ground-braking improvement to programming. The authors make a fair effort to compensate for this by making other parts of the paper as generally useful as possible. For example, they give a relatively comprehensive overview of encapsulation in other OO languages, and they derive a general catalog of requirements that should be fulfilled by a good encapsulation mechanism.

However, I think that this could be made more explicit/useful by restructuring the paper a bit so that the distinction between the general part and the C++-specific part gets even more clear. Currently, the discussion/evaluation of the encapsulation mechanisms of various OO languages is distributed over multiple sections (mostly 2 and 7, but also 4 and 1) and are inter-mixed with the problem statement and the requirements for the solution. I think that the paper would benefit a lot if there would be a clearer separation.

Also, I would suggest not only a disucssion of the various encapsulation features of the different languages, but also an evaluation that shows which of the requirements stated in section 4 are actually supported by the discussed languages (including a summarizing table). This would not only strengthen the suggested C++-solution (by showing that not many languages actually support such a flexible encapsulation mechanism), but it would also make the paper more useful for people who are not necessarily interested in the C++-specific part.

While the authors give an overview of encapsulation approaches in other OO languages, they do not really mention a lot about related work on a more scientific level. For example, I missed the discussion of object-based (rather than class-based) encapsulation in the context of this paper. While this concept is not (yet) supported by many mainstream languages, there has been a lot of research activity around this topic and it certainly seems relevant when talking about more fine-grained encapsulation mechanisms. As a reference, the authors could for example have a look at "Object-based Encpasulation for Object-oriented languages" (Schaerli, Black, Ducasse), which mentions quite a bit of the work in this area and also proposes a concept similar to "access by role" as proposed in this paper.

Third reviewer's review

>>> Summary of the submission <<<

The context of this paper is language engineering, more specifically C++ language engineering. The problem addressed by this paper is the addition of more sophisticated (Eiffel-like) efficient access control in C++ without changing the compiler. This goal was achieved by using template meta-programming in combination with the client passing an explicit argument such that at compile time (or even runtime for dynamic checks) it can be verifier whether or not the client has the right to use a particular method or class. The approach is illustrated with various usage scenarios.

>>> Evaluation <<<

The problem addressed by this paper fits the ECOOP conference and is interesting. The paper is well structured: it has a logic built-up and the authors try to be didactic. It has however to be noted that the quality of English could really be improved, and preferably a native speaker should have a look at it (several paragraphs throughout the paper are nearly incomprehensible). With some effort I was able to read and understand the complete paper, so this was not added as a point against the paper. Newer versions of the paper should be made clearer though. Regarding the structure I just had the feeling that the easier parts of access control in the beginning of the paper were slightly too long, while some of the template meta-programming tricks could be a bit better explained.

The core of the approach as I understood it is to add an extra parameter whenever invoking a method or using a class that has a controlled visibility. Note that this approach will work for any object-oriented language (and even beyond), from old languages like Smalltalk through Python, Ruby, Java, C++ or C#. The paper could therefore be made stronger than it is currently formulated without much effort. The current validation by using template meta-programming in C++ can just be kept without any problems. It is clear that somebody who would like to use the proposed approach in another language will have to use other language features to implement it, but that is no problem for the claim made.

What I did not like about the proposed solution is the overhead it imposes. The paper states a number of times that the approach requires a minimal syntactic overhead, but I found it to be actually quite heavy. Both the implementer’s side and the client side are impacted. I found it very hard to understand the access-controlled code and easily spot the protections it implemented. One could argue that this is the price to pay to get more sophisticated protection, but that (supposedly lightweight) solution then has to rely on a convention. This is where for me the approach fails to become truly interesting, and where I do not think that it will be used very much in practice. The key-based approach is more secure but even heavier.

Moreover I think that the problems mentioned in the previous paragraphs are not specific to C++, in which case one could argue that the realisation of the approach in C++ is feasible but not ideal from the syntactic point of view (I do like the fact that it is efficient though, which will be hard to achieve in other languages). But since the core of the approach is to add an argument and use that argument to implement access control, I think that it is the approach itself is quite heavy. Luckily we could always add the extra argument (for any method called), and then the approach would become much more interesting. This would require work on the compiler or VM (depending on the language), but the results would be much nicer. An extra (quite simple) pre-processing step for C++ would probably already be enough.

The part of the paper where more work is needed is in the related work section. Beta, Modula-3 and Jigsaw have features that can be used to do partial revelation, and that are of interest to this paper, as well a paper of Ecoop’04 on encapsulation policies. Note also that a number of references are not complete (4, 7, 8, 14 and 15), lacking years or publication venues. Throughout the text a number of references were put as ?, so possibly a number of papers did not make it into the references section. Definitely something to check and clean.

Some smaller remarks encountered while reading the paper:
- page 1, Introduction section, line 4: ’... possible services or messages the class offers to its clients.’ : messages should be methods here (a class can only offer methods).
- The next sentence ’Messages are specified...’ is incomprehensible, but messages are a runtime aspect and hence cannot be specified by a signature.
- page 2, line 5 and page 4, the description in Smalltalk: In Smalltalk instance variables are protected (methods in subclasses can access them), not private.
- Page 8, first paragraph of section 4: says that most results can be generalized to most of the statically typed modern OO languages: I think that

you do not have to limit yourself to statically typed languages, and that they also apply to dynamically typed languages.

- Page 9, end of first paragraph ('Thus no other possibility but defining namespace operators remains'): virtual methods and a common superclass do the trick, and are actually much more OO. The problems faced by C++ in this area are all due to the fact that it is actually a multi-paradigm language and that there are problems due to having both procedures (global functions) and methods (member functions) around, as well as base types (non-objects) and objects. This why some combinations work, and some don't. A proper explanation of this should be given instead of a number of examples.
- Page 9, string and char[] example: This example is specific for string and char[]. If a proper explanation of the underlying problem is given it is not needed (except to show a case where the system does some kind of mangling for you in a number of cases, like this one).
- Page 9, section 4.2: '...hard to argue why we would attach the operation to any of the classes as a member method.'. This is because operators (procedures, a non-OO concept) do not mix that well with objects. The OO solution is to have operators as methods, and then the confusion goes away. A method + taking one argument into account can be implemented on Matrix if one wants to be able to sum something with a matrix. Likewise for vector when you want it to be symmetrical. This is simple to argue: straightforward OO semantics.
- Page 10, first sentence: I do not agree with the remark on coupling. If you want to be able to sum vectors and matrices then conceptually they will be coupled, even when using a namespace function (removing the vector class will invalidate the global function). Worse, when the operation is implemented as a namespace function the coupling just becomes harder to see (but is there).
- Page 10, second paragraph: Java (nor C#, for that matter) is not a pure OO language. Base types are non-objects, and operators are, exactly like in C++, procedures (and not methods). The difference with C++ is that end-users cannot add procedures themselves (no operator overloading, which is the mechanism used in C++ to add procedures to user-defined types). Note that the + should be implemented on both Java classes (when associative one implementation can call the other to avoid code duplication). A double dispatch scheme can be used to avoid giving access to the private data.
- Page 12, section 5: I suppose that one could also use RTTI (runtime type identification) to achieve the same. This would be more costly but easier to implement.

Points In Favour

- Useful problem
- Possible solution formulated

Points Against

- Heavy solution that either depends on convention or becomes even heavier
- Related work section quite weak

Response (Optional)	Note: This (optional) response form is to be used *only* to make corrections to *factual* errors in reviews, if any, or to answer specific reviewer questions. The limit is 500 words! For better readability for the reviewers, please use newlines near the right side of the text box, instead of using your browser's automatic linefeed.
Count Words	

Submit Author Response

Note that feedback will appear in another window!
Responses consisting of more than 500 words will not be stored!

In case of problems, please contact Richard van de Stadt.

CyberChairPRO | Copyright © by Richard van de Stadt (Borbala Online Conference Services)