|  |
| --- |
| Circle Language Construct Drafts | Black Boxes |

## Black Boxes Construct Drafts Ideas

### Misc Issues

In previous projects the main articles about black boxing were finished, but other subjects were left unfinished. This article contains the unfinished material of some of these topics.

This article contains texts and loose ideas about the following topics:

*Deeper Exclusion*

A parent object’s restricting access to specific members of their children.

As said, the texts about these subjects are not finished yet.

#### Deeper Exclusion

Apart from controlling access to the type’s own procedures, a type can also control publicity of procedures *of its children*. That way you can publish a child, but make some of its public procedures inaccessible outside the parent. This way you can also exclude members of a base type. A parent can’t make a child’s procedure inaccessible to the child itself; it’s the child’s own procedures.

You can also exclude members of ancestors even further down the hierarchy.

Forms of exclusion other than Private are just called exclusion or privatising.

You can only make a procedure of a child *less* accesible, never *more* accessible. The parent can’t publish a child’s procedure that the child defines as Private. A child can’t put restrictions on how its members are privatised by the parent, so a parent has full control over the exclusion of the procedures its ancestors.

Exclusion access controllers are also called *access* *modifiers*, because they modify formerly set access controllers.

-----

Inheritance, Exclusion,

2008-10-15

This is also a blackboxing thing...

... but I put it in inheritance.

JJ

-----

Inheritance, Exclusion,

2008-10-15

Because it is complexity hiding, it is blackboxing.

JJ

-----

Exclusion,

20008-10-16

But it is just access control, so it is blackboxing, not inheritance.

You can apply access control combined with inheritance, to further specialize a class, object or system.

JJ

-----

Exclusion... so a grandparent imposing extra access control onto its grandchildren... Another advanced topic to cover at the end.

-----

Gee, Public and Private are already a form of exclusion, because the parent object is imposing restrictions to access to its sub-objects ( the references it contains. )

-----

When you make it private, it means that it is only accessible to friends...

hmmm...

-----

Deeper exclusion protects members of sub-objects, but how much can those sub-object members be protected, if they might also be referenced elsewhere. Somehow they have to be protected from being referenced elsewhere... fixed logical residence? What if you reference the object from elsewhere? Somehow it needs to be controlled, that an object is always referenced THROUGH the object that protects it... There is a link with user access control here, but I won't go there yet.

Yeah, there is a discrepancy between controlling access and the fact that

containment is now a volatile thing. Fixed containment needs to be

imposed in order to establish access control...

Multiple objects can impose another access modifier upon the same object,

so where are all these access modifiers stored? At first it will only be access modification of a direct reference, which is already under control of the parent object. But for deeper objects it is different... so I am not sure yet... are those deeper objects supposed to be exclusively contained by the grandparent object? Or can they also be referenced elsewhere, so that the grandparent has to remember the grandchild access modification settings and not just delegate that to the child, that holds the grandchild? I'm not sure. Perhaps details like that should be worked out when actually turning this language into an actual programming environment... and we should keep the focus on the general idea?

-----

Private, Public is exclusion

-----

< Public and Private are an application of exclusion. So then the base of the explanations can be only accessible and inaccessible. >

-----

Exclusion is the primary method for access control. Friends is the method to make exceptions to the privatisation imposed by exclusion. A procedure or type can make any other procedure or type they can reach a Friend, meaning that that procedure or that type has access to all members privately accessible in the befriended.

> And cover deeper access as well.

> Also cover how a fixed logical residence allows parents to impose a permanent, non-volatile access control. Qualified access to an object without it being a fixed logical residence does not guarantee access control imposed by the qualifiers. But if it is a fixed logical residence, the qualifier is mandatory and you can not go around the access control, that it imposes.

### Side-Issues

In previous projects the main articles about black boxing were finished, but other subjects were left unfinished. This article contains the unfinished material of some of these topics.

This article contains texts and loose ideas about the following topics:

*Inclusion*

This topic may be cut.

As said, the texts about these subjects are not finished yet.

#### Inclusion

Apart from a *decisive* access modifier, you can also give a procedure a *suggested* access modifier. This means that when you put an object in a type, a procedure of the object will get its *suggested* access modifier, for instance Private, but if its *decisive* access controller is Public, then you can still upgrade it to Public. This is called *inclusion*. The suggested access modifier is always set to less accessible than the decisive one. A type can also give suggested access modifiers to procedures of deeper objects.

-----

> 2009-07-06: Inclusion might only be useful for programmers. But programmers get automatic friend access, so real, exclusive access control is already the suggestion and everything can still be included, so it might be an unuseful concept.

### May Cover Last

In previous projects the main articles about black boxing were finished, but other subjects were left unfinished. This article contains some unfinished material. The topics are considered of lesser importance, compared to even other topics, that were not finished.

Some of the topics might be deleted and others might not.

#### Level Limitation

< Exclusion is restriction set by a container. Level restriction is like exclusion set by a child *for* the container. >

-----

> 2009-06-27: I am not sure why I would need a child object to restrict the number of levels up it is accessible… why? for protection? Well… I would better resort to use access control for such protections…

-----

Basic access controllers are decisive, but level limitation adds additional restriction.

Level limitation is paired with the basic access controller.

Pairing object level limitation with the Public access modifier, makes a procedure public only to limited number of levels up the ancestry.

Public *Up 2*

A procedure with this access modifier is accessible but two levels up the ancestry. Outside those levels, the procedure is inaccessible.

|  |  |  |
| --- | --- | --- |
| No object level limitation: Public | | |
|  | ≈ |  |
| Object Level Limitation: Public Up 2 | | |
|  | ≈ |  |

Level limitation works the same for the Protected access modifier, but gets the additional restriction that comes with that access modifier:

Protected *Up 2*

A procedure with this access modifier is accessible two levels up the ancestry, but inaccessible outside the first circular descendant.

|  |  |  |
| --- | --- | --- |
| Protected Up 2, contained in three triangles | | |
|  | ≈ |  |
| Protected Up 2, contained in a triangle contained in two circles | | |
|  | ≈ |  |
| Protected Up 2, contained in three circles | | |
|  | ≈ |  |

Level limitation can also be applied to deeper exclusion than **Public** and **Protected**. So a parent can impose level limitation on a child procedure.

|  |  |  |
| --- | --- | --- |
| Level limitation on deeper exclusion | | |
|  | ≈ |  |

You can also apply level limitation for on symbols shalower than the target procedure.

|  |  |  |
| --- | --- | --- |
| Level limitation on deeper exclusion | | |
|  | ≈ |  |

Level limitation is defined there where it doesn’t have effect yet. It only has effect higher in the ancestry. Basic exclusion is always defined there where it has effect. So level limitation automatically excludes members at *n* levels up the ancestry.

< Figure out a better diagram notation. >

Maybe call it Level Restriction.

#### Globality Level Limitation

<Pictures>

Apart from object level limitation there is also globality level limitation.

Public *Globality Up 1*

When an access modifier is paired with a globality level limitation, the access modifier works normally *n* level up the *globality ancestry*. But for higher globalities the member is inaccessible.

|  |  |  |
| --- | --- | --- |
| Public | | |
|  | ≈ |  |
| Public Globality Up 1 | | |
|  | ≈ |  |
| Public Globality Up 2 | | |
|  | ≈ |  |

Protected Globality Up N works the same as Public, but adds the extra restriction that comes with protected.

> 2009-06-27: Globality level limitation is that VB6 Friend idea. That is probably why I also introduced object level limitation to begin with: as a bridge towards the concept globality level limitation. It is more like overkill… I do not know yet how VB6 Friend access is supposed to look, but perhaps you’d better just appoint inside which globality you still have access to it or something, instead of coming up with Object level limitation and only to build globality level limitation only to have a substitute for VB6 friend access control.

> 2009-06-29: But level limitation and then globality level limitation is a lot of systematics to introduce just to facilitate a VB Friend access modifier.

<Pictures with that? >

<‘Globality Up 2’ is too long, because Public Globality Up 1 will be used commonly. Visual Basic’s Friend access modifier is the same as Public Globality Up 1. I might need to invent a specific term for Public Globality Up 1. I won’t use ‘Friend’, because that would conflict with >

#### Downsides to Black Boxes

Data protection does not really have a downside.

Complexity hiding, though has a downside.

Complexity hiding does has benefits: it allows you to focus on overview, focus on the main point of a system, object or command.

In professional software development it is a blessing, that you get access to a wealth of objects that provide functionality for you while you do not need to know or see exactly how it works inside. It allows you to focus on the main point of the program. If you know how to properly use them, there is no downside.

The downside of complexity hiding, though, is that you do not know what is going on inside the object you are calling. You may think something you call upon is a simple procedure, while in reality you are consuming so much the computer’s power that it makes your application perform really poorly and probably puts a break on your whole computer’s performance. You might not want to know all the details, but when something is abstracted you do not know exactly what is done.

There are methods to overcome this though. It is not concrete yet, but the inner complexity of an object or command could be expressed by size. That will give you a relative estimation of the complexity of what you are calling upon.

< … >

#### Get For Access and Get For Copy

Access can be controlled for regular procedures this way. You can also control access to system procedures. When you make an object Private, you’re in fact making its Object Get procedure private (which can even get different separate access controllers for Access and Copy, ~~Run Time~~ Copy ~~and Design Time Copy~~).

> 2009-06-29: The difference between design-time and run-time is gone in the new language. There is only differentiation between program authors and users through user access control.

-----

There might be a separation between access controlling Get for Access and Get for Copy... but not much more. I used to have a difference between Run Time Copy and Design Time Copy, but that is bull now, because there is not difference between run time and design time. There may be a difference between different types of users: authors and users... but that is user access control, which is not covered here.

-----

<There are also situations in which you want to disallow copying an object reference to an argument, but only allow consult of the argument. (that’s access control of system procedures, actually)>

#### Inaccessible System Aspects

What if you want an aspect to be inaccessible to even friend objects. Then the only one able to access those aspects is a programmer… That is strange… giving nothing in the system access to something, but still the programmer can access it. That is weird.

Let’s stick to the plan where either everything is accessible or a selection of thing is accessible.

#### Compared to Traditional Black Boxing

I already did compare it too much to other systems, but for a proper explanation it is important to draw that comparison to regular OO. So in that case it is allowed to draw a comparison (it is a rule not to go on and on comparing the new language to other systems…)

> Deals with the two differences to traditional:

- Not only commands are friend with their object

- Containment and referential interchange.

> This overlaps with issues put under *Notation*.

-----

Public & Private,

2008-09-15

**Private** means on the inside, **Public** means from the outside.

JJ

-----

Public & Private,

2008-09-17

So Public does not means that you automatically have Private access as well.

JJ

-----

> 2009-06-26: The last two ideas do not apply anymore. Public and Private will not have anything to do with outside or inside… Privates are accessible to friends… whether those friends are on the outside or on the inside.

Perhaps you could also work out the traditional implementation of public and private. In that setting the original notation is usable. Now it is all changing. Perhaps it will create clarity if you isolate the original idea.

It is superseded by the Friendship idea. And later, the Friendship idea will probably be superseded by Access Control: public and private and user access control are the same concept. But you have to make dues with intermediate solutions. So the traditional idea behind Public and Private is also important.... maybe.

-----

The main problem with the original idea about Public and Private, is that a programmer does not define a containment structure, and there is only a referential structure... Something contained inside another thing can easily also

be referenced elsewhere. This counts for methods too, so they are not exclusively part of one single object / class.

This creates problems for the traditional approach of Public and Private.

-----

Who knows. Maybe the whole idea I have now about the Black Box principle will totally change in this project.

The main problem I still see, is that in the new system, containment is not so actively set by programmers anymore, because you are mainly working with a referential structure, which makes the containment structure more volatile.

It is not so much a Private member’s being protecting *from the outside* anymore, which is the traditional view on black boxing. It is about a Private member’s being protected inside its circle of friends and protected from non-friends.

Outside the circle of friends the black boxed contents are not even visible. That’s the way black boxing is still directly depicted in the diagram.

However, a member is made private, accessed from its container. So it is a matter of inside-outside the container (the container, that makes its member private). But the private member is not accessible only from within the container, it is accessible from the outside to Friends of the container.

So there is a slight shift in the view on Black Boxing.

But you can have two views in diagrams: a friend accesses a private member going from outside to inside, but you can also have the friend displayed inside the befriended, accessing the private members referencing from inside the friend to outside the friend, but inside the befriended.

-----

Comparison to traditional black boxing,

2009-07-01

Hoe komt het black box principe nog naar voren? Wordt dat black box principe letterlijk genoeg zichtbaar?

It is so incredibly referential. Euwch.

The thing missing is the ‘accessed on the inside’ aspect, that in traditional black boxing was there. This is because in traditional OO you do not see that you are accessing a this parameter when you are accessing members of this. So it seems you are accessing things directly, while really you are accessing the members of the *this* object.

But I sort of miss the idea. Perhaps you’d want to get it back. This will only be possible, if you do not express access to this members as an access to a member of the argument, but as an access to a member of the parent object. So then instead of referentially displaying a this-argument inside the command, and a command inside the object next to it, you have to display a command in an object and …

But this is more for automatic containment (or relations). This is the discrepancy between automatic containment and displaying bidirectional relations, not to be covered here. But it is to be emphasized here. Perhaps you do need to show the less-referential approach, and the inversion of it and also show the referential view. Both notations. Because black boxing is about containment… and containment is a little lost.

For the time being this all will be put under ‘Comparison to Traditional Black Boxing’, which is not called a *main issue* for nothing.

How stupid is the older notation of public and private when you use it in the less-referential containment notation?

I have to make dues with the fact, that there are two notations: one with one symbol displayed inside the other with the reverse relations implied, and a notation with one symbol displayed inside the other, but the inverted containment right next to it, and them being connected together.

JJ

-----

A command’s access to a parameter’s privates or only its publics.

-----

< Perhaps save this for the ‘traditional black boxing’ section: >

A typical situation of friendship in object oriented programming is where an object has private and public data, and there are friend commands. The commands can access the private data of the object, thus becoming the protectors of the object’s private data.

This is a typical situation in traditional object oriented programming: class methods. In the new computer language it is a little different: instead of having a class with methods in it, there is a class or object with friend methods, that are independent entities, which have access to the private contents of the befriended object.

-----

2008-05-18

Access to the private contents of a class creates a tighter bond between a command and a class. This concept is important.

JJ

-----

Public & Private,

Friend Access,

2008-09-22

(out of the original document Commands & Classes Loosely Coupled)

In the new computer language’s implementation of parameters, commands only become part of a class if the class is used for one of the parameters of the command.

This makes it possible for a command to become part of multiple classes.

This is a richer and more dynamic approach, than object oriented programming languages. You may now think you are losing control over the system, if a class gets so easily extended with more commands. But you still have control, as will be explained in the Interfaces article group.

There is one thing however, that’s missing from the story. In C++ and other object oriented programming languages, there was one parameter of a global method, that was the this-parameter. The class of the this parameter determined which class the method would be part of. That parameter was the only parameter in the method, whose private contents could be directly acccessed by the method. This is the one thing missing in the story of Parameters: when does a command have access to the private contents of a parameter? The answer lies in Friend-access. A command has access to the private contents of a class, when the class declares its command to be Friend. When a related object is declared Friend, then the related object can access the private contents of the one that declared it Friend.

Commands do not decide on their own whether they have access to a class’s private contents. A class grants a command access to its private contents.

This makes the command *more* part of the class, but a command is still a separate entity, that can apply to multiple classes.

Private members of a class are useless, unless there are commands that use these private contents. So the informal connection between classes and the commands, that is established by the class’s granting the command access to its private contents, is very tight, and those commands are actually the protectors of that class’s private contents.

Classes giving commands access to their private contents formally is a form of access control. But informally the commands, that have access to the private contents of a class, are the protectors of that class. The class appoints these protectors. Commands do not decide on their own to become such protectors.

JJ

##### Brainstorm

In traditional black boxing in object oriented programming an object had commands and each private member of that object was accessible inside that command.

In the new computer language there are the following differences:

- A command is not part of a specific object or class

- A command and an object are basically the same thing

The main idea for the solution is Friendship.

…

If the friend object is a command, then if the command has a reference to that object, from within the command, you can reference anything private inside the friend object. This is analogous to the this-argument of a command. The private contents of an object passed as the this argument can be accessed from within the command. However, in the new computer language a command can have multiple this-arguments: befriended objects.

So friend commands are like class methods.

< state that every command with a parameter with a class becomes that class’s method, but not as tight a bond as a friend method. >

In traditional black boxing it only applies to friend commands (class methods) and their befriended objects (this-arguments).

But in the new computer language you can also have:

- Friend objects of befriended commands and

- Friend objects of befriended objects

- Friend commands of befriended commands

In the first situation an object can reference access the private contents of a command. That sucks from an object oriented programming point of view, because this does not protect the command’s private variables and the command’s procedure definition. But that is just what friends are about: they ARE protectors of an object’s private members. They have full access to it, and take responsibility of it. That is also what class methods do in traditional object oriented programming.

In the second situation one object can access the private contents of another object. That is *exactly* the same as the C++ implementation of friend classes.

Actually, class methods & C++ friend methods are the exact same concept in the new computer language.

> 2009-07-06 Actually, they are not. Friend in the new computer language applies to a specific relation counterpart, while friend in C++ applies to any relation between objects of class A and an object of class B.

In the third situation a command gets access to another command’s private contents, making the command’s local variables and definition accessible to the other command.

I am not sure what you are supposed to do with getting access to the private members of a command… but in the simple setting it should be possible. However, to prevent circular creation of command calls on (indirect) recursive calls, in the commands topic it was decided to only create a command’s private contents, just when it is running. So in that setting there is no point to the situation where a command gets befriended.

##### Other Brainstorm Items

The complexity hiding aspect is clearly expresses the black boxing concept in the diagram. It is the hiding aspect that makes the black boxing, not the inside-or-outside aspect.

##### Using command symbols

< Cover ~all (?) the notations when one of them is a command symbol? It is to point out how it would look in the traditional approach where commands are friends with their class… >

-----

2009-07-20,

< Access controlling executing a command is inadequately gone into yet. >

-----

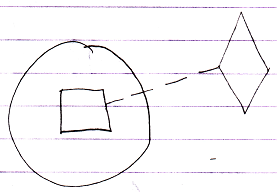
< Perhaps these specifics to commands must be separately adressed, completely separately. You also need to introduce access controllling Use As Class / Use As Definition in use with commands. >

-----

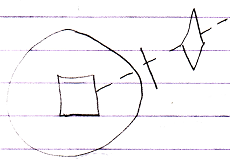
The essence of access controlling the Execute aspect is basically to make a command Private: the command can only be executed from Friend objects.

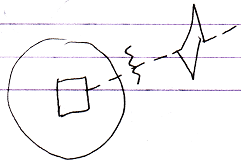
You have to consider how the effect of accessing a system aspect looks, and then base the access connector on it.

The effect:



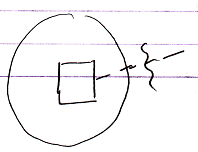
The access connectors:





The problem with that is, that access controlling the Execute aspect is about the executability of the command object itself, not about the usability as a definition, and for the call to be executable.

Usability as a definition is expressed through access controlling Use As Class, anyway:



So that makes you able to access control being able to call a command, so basically to make a command Public or Private for calling.

JJ

#### Other Details

Friends can always pass a reference to a private object after all. Such 'betrayal' by friends is allowed. The friends are the protectors. If they choose not to protect, than that's THEIR choice. No problem.

> 2009-07-02 It’s a detail.

-----

Fixed containment,

But it is friends that are supposed to be protectors? And fixed containment, also makes the container a protector. So are those friends as well? The protected can only be accessed THROUGH the friend... Hmmm... I have diagrams floating around in my head, but I cannot hack it yet. I can't see the class method (methods that are friends of the object) as the protector, and the class itself as the protector of its the items it contains.

Gee... I just do not have it figured out yet.

JJ

-----

Detail: mutual reference is required for a friendship to even work.

-----

Consider the situation:

access to a selection of private members especially selected for the function.

-----

Make an object befriended to a command.

Or befriend to a specific reference to a command.

-----

If you can object-protect the data aspect and the parent applies exclusion to the data aspect, then you have it, that data is accessible on the inside, but not on the outside… do you all of a sudden have that literal inside-outside access control back then?

### Might Not Cover

These topics are considered of such less importance, that they eventually might be deleted. This is because they might introduce discussion where no discussion is required. And that introduces confusion.

#### Access Modifier Get & Set

Interesting new system commands:

- Access Modifier Get

- Access Modifier Set

So would each system commands have its own system command

controlling the access modifier?

But what modifies the access to the access modifier system commands?

> Details...

##### Changing Access Modifiers at Run Time

Being able to change access modifiers at run time may be interesting. That way you can for instance change the interface of an object dependent on its settings. Changing a procedure’s access modifier is done by calling the procedure’s Access Modifier Set system procedure. There’s also an Access Modifier Get procedure to obtain the access modifier. This adds two system procedures to a square:

Access Modifier:

Get and Set

The Access Modifier property procedures can be access controlled themselves, so that they can be made inaccessible, for instance when you don’t want to be able to change the access modifier at run time. The access modifiers of the Access Modifier property can’t be changed at run time themselves. In other words: the Access Controller property procedures don’t have an Access Controller property of their own. This complies with the rule: system procedures don’t have system procedures of their own.

A single procedure can have several access modifiers so there are multiple access modifiers to get and set. This would suggest that the Accesss Modifier property has a dimension. However, when you call the Access Modifier property, you return or set the access modifier imposed by the object you’re calling from. The object you’re calling from is a hidden argument to the Access Modifier property. The access modifiers aren’t really stored inside the procedure, but in the object symbol that imposes it.

If the access modifier isn’t variable at run time, then the type defines a fixed access modifier. If the access modifier *is* variable at run time then the access modifier can be different for each instance of the type.

##### Changing ‘can be’ class, interface or object

System commands for changing access control to ‘can be’ class, interface or object.

#### Grouped Access Control

Using triangles to group members with equal access controllers. As with exclusion, the access controller of the triangle limits access again. It won’t make things *more* accessible.

#### Conditional Access Control

Replace the protected keyword by a concept like ‘when being a triangle’:

Basically you can optionally totally redefine the access for the case when the type is a used as a triangle.

<When being a globality?>

Or perhaps even make it as general as ‘conditional access control’: Public if triangle.

Or perhaps I am looking too much for systematics behind it.

Perhaps the base of Use-As access control should (eventually) be based on conditional access control. But for now it can be limited to the conceptual explanation of the most usable things.

#### Delayed Creation of Object’s Privates

Public & Private,

2008-08-16

Perhaps for objects, private contents also only need to be created, when one of its friend commands is run.

>   
But this is such an influential implication, that it needs to be covered elsewhere. Privates only make sense, when only a selection of commands, can access those privates. So only when an object’s friend command is executed, the object’s private contents are needed. But a sub-object’s system commands can also access privates…

JJ

#### Access Symbols And Accesses

< Just find an appropriate time to mention that each access symbol represents an access, but that it’s also used to denote direction, simply because direction can be read from it >

-----

2009-06-28

It was realized too little, that each line crossing is an access...

but in the automatic containment setting it is not. The crossings of imaginary reference lines are not accesses. The crossings of lines going from the imaginary reference to the logical residence... I am not sure if those are active accesses. They could be. But this is more like an implementation detail.... Not important enough.

JJ

-----

Maybe it's a good rule to only show access symbols to denote:

- Direction

- Exclusion of a procedure

- Denote explicitly the publicity

-----

2009-06-28

Each line that is crossed is a separate access. That is probably why I wanted to have those extra ticks in expressing access modifiers, but access modifiers and actual connections are two different things. So the syntax of those two things is a different deal. Whatever

JJ

#### Access Controller Parts

In one access controller you can use an exclusion access controller, an object level restriction *and* a globality level restriction:

* Protected Up 1 Globality Up 2

Access controllers as such are not usual, though and *if* you use them, they’re not very diverse.

-----

> 2009-06-29: So basically this brainstorm item is about how an access controller can consist of different elements: each aspect is represented only once.

#### Uses of Access Controlling System Aspects

An object symbol’s Object Get procedure must be accessible to make you able to access its sub-object.

Exclusion of Set procedures controls who can set a symbol’s lines. Exclusion of Get procedures controls who can get information about lines or who can access the object.

> 2009-07-06: Or who can use the object as line source.

#### Other Details

< Give a story about the true meaning of an access symbol that denotes direction. That it is a syntactically correct access symbol. Access symbols can used to denote direction, because access symbols *can* point out direction.>

-----

< A procedure’s primary access controller is only made Inaccessible to ‘outcomment’ code. This is a rather small concept. The rest of the use of Inaccessible and Accessible is in exclusion. Therefore, I need to denote in short that this is the only use of the primary access controller. I can basically always speak of access modifiers when talking about Accessible and Inaccessible. I can use the term exclusion in a lot of places where I now use ‘access control’. I also need to put the first section ‘Accessible and Inaccessible’ in the exclusion section. >

-----

> 2009-06-26: I do not even know what I mean by this: You will not be able to totally make clear what private and friend means in this context, but you can just use the term command anyway, to make it clear.

-----

< The word access modification can actually be replaced by ‘exclusion’ in many places, but not everywhere by far. >

-----

Detail:

The symbols used here are actualy access symbols, which I’ve already introduced. Access symbols can denote direction, and they also denote where access takes place, but they’re also used to denote access control.

Interesting Applications of Exclusion

So examples of interesting applications of exclusion are:

* Making procedures Public or Private
* Excluding procedures of a public object
* Excluding procedures of a base object
* A parent’s controlling *access to objects*

-----

Access to *procedures* is controlled. But access control usually defines which *objects* get access to them. In a special case you can control which specific *procedures* get access.