|  |
| --- |
| Circle Language Spec: Black Boxes |

## Black Box Details to 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 >

#### Access Controlling Globals

> This issue is more important than level limitation and globality level limitation, but those two concepts are used in the implementation as laid out below (the implementation will probably change in the future.

Access modifiers in Globalities:

* Global Inaccessible
* Global Private
* Global Public

To understand what happens when procedures inside globalities are given access controllers you need to view the globality as an object, that is referenced from any of its ancestor objects:

|  |  |  |
| --- | --- | --- |
|  | ≈ |  |

If a global procedure is Inaccessible, it’s just not accessible at all (*‘outcommented’*), so not really global either. If a global procedure is Private, it’s not really global either, only accessible by the globality object itself, just like something private of a circle object. If a global procedure is Public, then it *is* global. It is then accessible from any ancestor within the globality.

When a global procedure is Public it is however also accessible *outside* the globality, as a public procedure of the globality object. For a procedure to be global inside the globality, but not accessible outside the globality you use globality level limitation: **Public Globality Up 1**.

> 2009-06-27: And here I am attempting to differentiate between VB6 Friend on a global level and VB6 Public on a global level. I am trying too hard to make a match between access modifiers I know from VB6 and C++ and match them with a new language. I was too afraid to reinvent each atypical concept, and insisted to put those concepts inside a single systematic. That is why I went to such an extent with this level limitation stuff. Ok, I can respect it now. It is an attempt to put those access modifier situations into context, but I think this should only be an intermediate implementation of it and in the near future an easier approach needs to be come up with. This stuff here should be seen as just a brainstorm.

It is not a permitted to make a global procedure Protected. A pentagon is not exchangable with a triangle, so Global Protected wouldn’t have much meaning.

> 2009-06-29 That, by the way, is not true anymore. You can reference a globality triangularly if you want.

Access Modifiers in Globalities needs to be further thought through. It gets tricky as you put it in diagram code.

#### Simplified Access Connectors

Could also be called: Access Control Presets.

< Wat wel zo is is dat de standaard access control van system procedures er voor zorgt dat je niet *zo maar* rare dingen kan gaan doen. Erg belangrijk als je effe snel iets programmeerd, dat je de echt rare effecten uitsluit zonder extra moeite. >

< 2009-06-29 Bedoelde ik hier oorspronkelijk mee de macro-keywords voor access control? Zoals bijvoorbeeld Public Read-Only Value ofzo? >

<

I don’t have a special notation for publicity of Set procedures.

Set public or not is kind of like read only or not. Maybe I need a read only notation.

Write only is also common, though. For instance of procedure reference input parameters.

>

<Also cover constants>

When you give a system procedure an access controller, access to the procedure or object represented may get a special name and diagram notation.

An object symbol with an accessible State *Read* procedure, but no State *Write*, is called a Read Only Object.

An object symbol with only a Type Read, can be considered just a Type. Not an object or interface.

An object symbol with only an Interface Read, can be considered just an Interface. Not an object or type.

…

Etcetera, work it all out

Standaard access controllers

* Standaard zijn Add en Remove geblokkeerd. Zo zijn er nog meer dingen standaard geblokeerd

De klasse kan zelf initial access controllers aannemen die je ook weer kunt veranderen.

Something is constant if it can be set by programmers , but not by users.

< By default everything is public. When studying the access configuration of a symbol you might only want to see what differs from the default >

<The standard access modifiers for system procedures of procedure symbols directly inside a procedure symbol are different from procedures directly inside object symbols. For instance, Symbol Get for Reference of a clause is Private by default. Symbol Get for Reference for an object member is Public by default>

A call is by default private.

The access controllers of the procedure’s sub objects determine which purpose they serve. Maybe I should write a section about that in Access Control.

Required and Optional parameters as well as In, Out and Thru need to be rediscussed in Access Control, because it may be so that it is required for you to Object Set a parameter, while the procedure changes the state of the target

object. This is like thru too, but a totally different fashion and there are more ways of throughput.

If a configuration of any sort is a common one, it might be nice to make something like a typedef so you give the configuration a name of its own, Like if you commonly use ‘Public Up 1 Globality Up 2’, you might want to give it a different name.

Consider combining the concepts Presets, Macro-Keywords and just a brainstorm about nice-to-have keywords for common access control situations.

Best example: Public Read-Only Value. Read-Only Value.

You do not even have to introduce completely sound systematics, you can also just shed light on the general concept.

Those 'macro' access modifier keywords are not very important.

There may be an analogy between macro keywords and a C++ typedef.

(perhaps only for names)

( I think this issue was also temporarily put in the Parameters chapter originally.)

But what about ‘making a command Private’? What do you mean by that? You can access control the Execute system aspect of commands, but that does not make other aspects of a command object private, such as the Object Set aspect (turning the command symbol into a command reference).

Should there not be something implied when a connector is not there: in a friend relation this will create a lot of connectors… so no connectors should mean everything is accessible in a friend or one connector should be the placeholder for any other connector… or some common connectors and others (such as Class Get implied )

In definitions, publics could also be differentiated from privates by privates not getting access connectors? Well… friends can access them, so they need access connectors…

When you have to display a lot of connectors, you might summarize them, and when you float over them, they expand into options.

Maybe first you have an object connection line without a connector. You float over it, it splits up into an class, interface, object and value line, you float over one of those, it splits up into for instance Object Get and Object Set. I know that does not cover all the options, but it might be an idea to display the connector summary and fan it out when you float over it.

Explicit display that something is private

is optional.

Explicit display, that something is public

could be made mandatory, because

is it not overhead in notation: it clearly

indicates a connection point.

If you do not see any access connectors, do you assume, that everything is accessible,

or that nothing is accessible?

That is a good question for the display of access connectors.

But not essential to cover.

Questions:

- Can you make it an option to only show befriended commands inside an object, instead of friend and non-friend commands.

I am not sure yet. It is not clear.

Another fact:

One of the most important things to access control is an object’s being accessible at all or having read-only access to it. Another important access controlling capability is to only be able to retrieve (and set) the Value of an object. Perhaps, if you state that Set is public it automatically means, that Get is also public, and if you state that Get is public, it automatically imples that Set is private.

> Perhaps it is interesting when one user writes a value, but other users can not read it out, but that is sort of outside the scope: that is user access control. System access control is covered here.

Perhaps that way the number of possible access controllers becomes more limited, so you do not have to display each possible access connector: Object Get implies any kind of connector.

Access controlling for just allowing a Class Get is stupid as well: who wants that? Who wants to only be able to access the reflective data of an object, but not the object itself?

The notation is simply drawing the symbol a dotted line or a dashed line,

which makes it only usable as an interface or class.

How you should express something only being usable

as a class but not as an interface is not clear to me.

I do not think I really need that and I do not think I

need to clear that up.

But whatever, this is for later.

Also useless: allowing Set but not allowing Get… if you can already write it, why not allow reading it? Is there any point to that? The important thing is write-protecting, or not access at all… but read-protecting, while you could write to it? It seems unlogical.

That seems a whole lot of Get and Set purposes. In practice, the set up of a symbol’s system procedures is not that complicated. There’s a standard setup: the most usual system procedure setup. Only deviations from the standard are additionally denoted.

#### Downsides to Black Boxing

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 loosing 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 analogus 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,

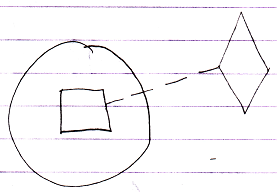
< Access controlling executing a command is inadequately gone into yet. >

< 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. >

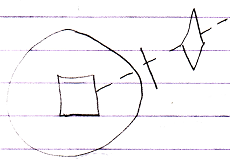
< The essence of access controlling the Execute aspect is basically to make a command Private: the command can only be executed from Friend objects.

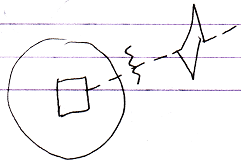
You have to consider how the effect of accessing a system aspect looks, and then base the access connector on it.

The effect:



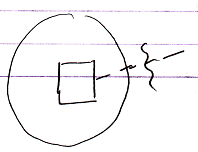
The access connectors:





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 can not 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?