|  |
| --- |
| Circle Language Spec: Commands |

## Creation Behavior of Commands

### Creation Behavior of Calls

A call can be present inside an object or inside another command. When a call is created, it is not immediately run, so that you get a chance to set its parameters. Before a command call is run, the command call’s public contents are there: its parameters. Those parameters are copied out of the call’s definition. The private contents are not there yet. A command’s private contents include private objects, clauses and command calls. The private contents will be copied out of the command defininition, only just before the command call runs. The public contents of a command call are added when the command object is created. The private contents of a command call are only added when its about to run. So the general rule is: creation of private contents of a command call is delayed until just before a command call is run.

The reasons for the delay of creation of private contents are explained later. First, the steps of a command call’s creation are laid out one by one.

Before a command is running, it is in the current state:

- Command is created

- Parameters / public contents are present

- Private contents are *not* created  
(private objects, clauses and command calls)

- Reference and Object Out parameters are already assigned

- Value parameter assignments refer to their sources and targets

Right before a command executes:

- Value In parameter assignments are executed

- Private contents are created  
(private objects, clauses and command calls)  
(copied out of the command’s definition)

- *Sub*-commands’ parameters / public contents are created  
(copied out of the sub-command’s definition)

- Mind, that the sub-command’s private contents are not created, which disincludes its private objects, command calls and clauses.

- *Sub*-commands’ Reference and Object Out parameters are assigned  
(targets copied from the parent command’s definition)

- Sub-commands’ Value parameter assignments refer to their sources and targets  
(copied out of the parent command’s definition)

Then the command runs, which means it runs all its sub-commands one by one, following the same procedure of call creation.

After a command call has finished:

- Value Out parameter assignments are executed

- Private contents are released  
(private objects, clauses and command calls)

#### Calls In A Parent Command

A parent command, that executes, automatically executes its sub-command-calls one by one. This means, that when a parent command is about to run, the sub-command-calls are created with *public* contents only. A sub-command-call’s *private* contents are created just before a sub-command-call is run. After a sub-command-call has run, the private contents are released, but its public contents remain. Sub-command-calls are the parent command’s private contents, so after the parent command is done, the sub-command-calls with their public contents are released. Because the parent command has a sole reference to its sub-command-call, releasing the sub-command-call means, that the sub-command-call is destroyed. But the parameter objects of the sub-command could still keep existing if they are still referenced in other places.

#### Original Problem & Solution

This was the original problem:

When a sub-command would only be created just before it is run, there is a problem: if a sub-command is created only just before it is run, when can a parent command fill in the parameters of the *not yet created* sub-command? Parameter passings could not reference the sub-command’s parameters. That’s why before running the parent command, the sub-commands need to be created.

But when creating the sub-command even before it is run, there is another problem: if sub-commands need to be created when their parent is created, the sub-commands of sub-commands would also need to be created, and their sub-commands and so on. So the whole call structure would have to be created before the parent command could even run. This could even have resulted in circularities, that would make the system hang.

The solution was, to create a call before it is run, but only the public contents, so that a parent can set the parameters. But the private contents are only created just before the command is run, which creates its sub-commands, but again only its parameters. This prevents recursive creation of the call structure, and gives command creation a neat and steady pulse. So this all makes it doable. Otherwise there could have been an endless recursive creation procedure, before any command could ever run.

#### Problems Solved By Delayed Creation

The creation behavior of command calls also solves *when* and *when not* to display a command’s private contents.

The problems solved by delayed creation of a command’s private contents are covered in separate articles:

- *No Overhead of Command Creation*

- *No Circular Command Creation*

- *No Private Contents in Calls in Definitions*

- *Calls in Calls Show Privates When Running*

#### Delayed Creation Of Private Contents Only Counts For Command Calls

Delayed creation of private contents only counts for command calls. Command calls redirect their definition.

But if a command object does not have a definition, then it defines its own definition. For command objects that define their own definition, private contents have to be created all the time, because nothing else defines its private contents but the object itself. This counts for command definitions. This also counts for *active* command definitions, which are executable object that define their own definition. But it also counts for clauses. See the article *Creation Behavior of Clauses*.

#### Compared To CPU-Like Call

There was a lot of brainstorming about how this relates to the way it goes in other programming languages, which have a command call creation scheme totally compliant to the CPU’s way of handling command calls, which is not in the spirit of *commands as an object*. But that brainstorming was postponed, because it did not have anything to do with a goal, but more with optimization of the new computer language. The contemplations are still there in the article *Comparison to CPU-Like Call Instantiation*, as an unfinished brainstorm, that may be later worked out, to better take advantage of CPU power.

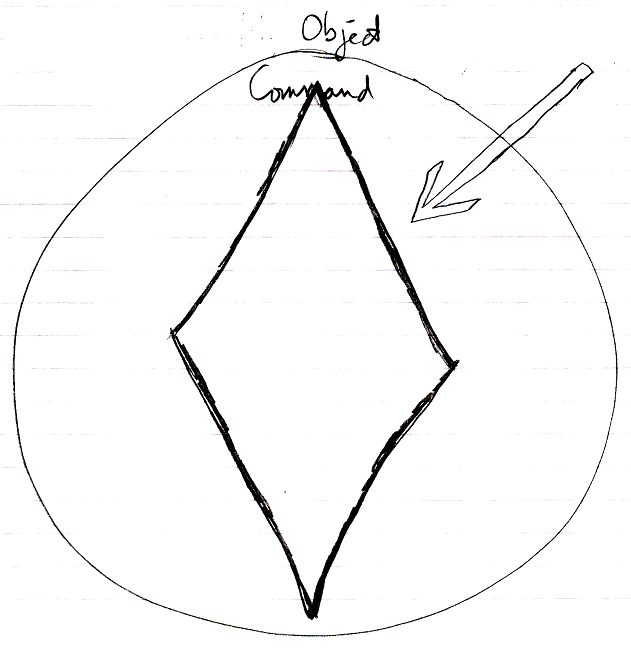
#### Diagram Notation

The creation behavior of calls was already explained in the article *Creation Behavior Of Calls*. The current article demonstrates the concept in a diagram.

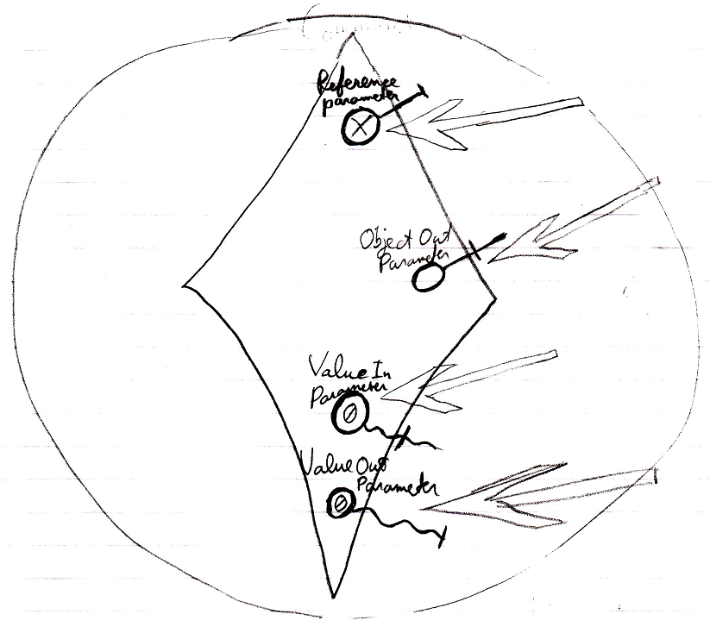
The steps of creation of a call are clarified with diagrams.

Before a command is running, it is in the current state:

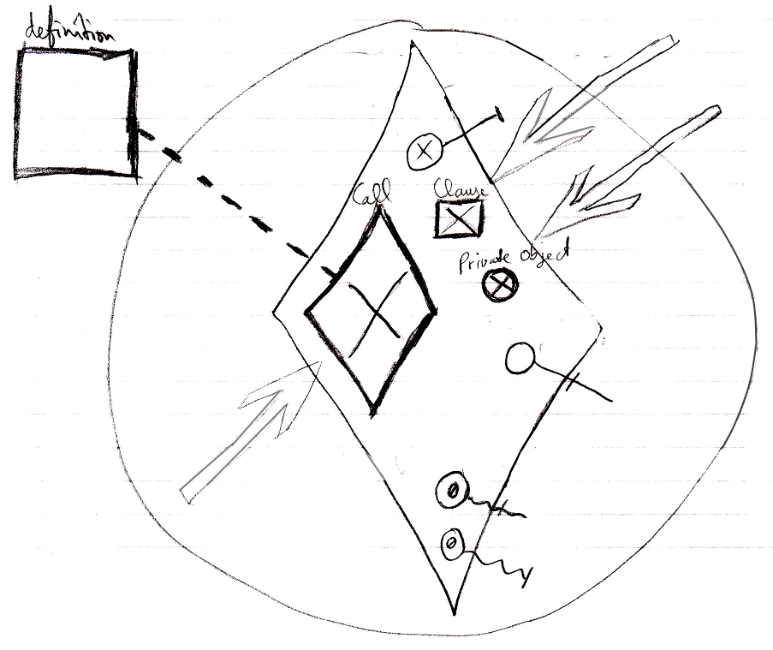
- Command is created



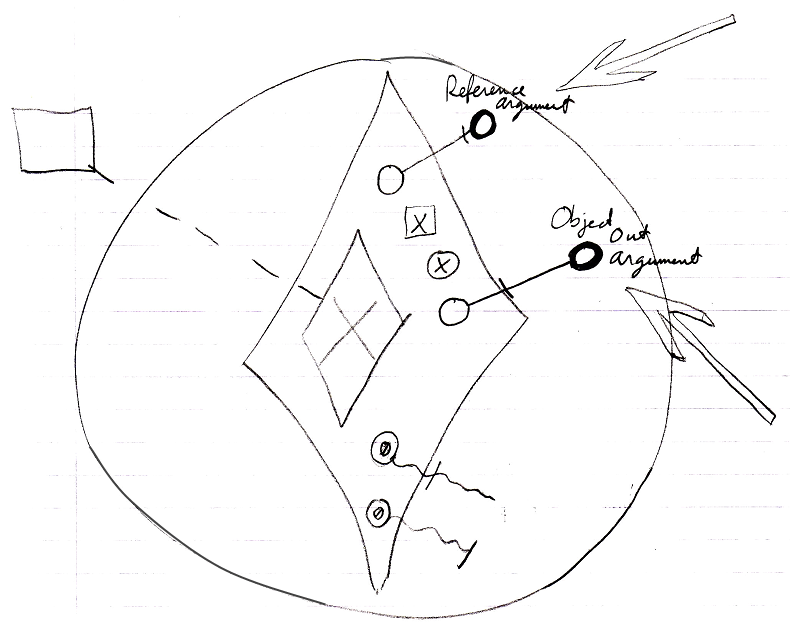
- Parameters / public contents are present



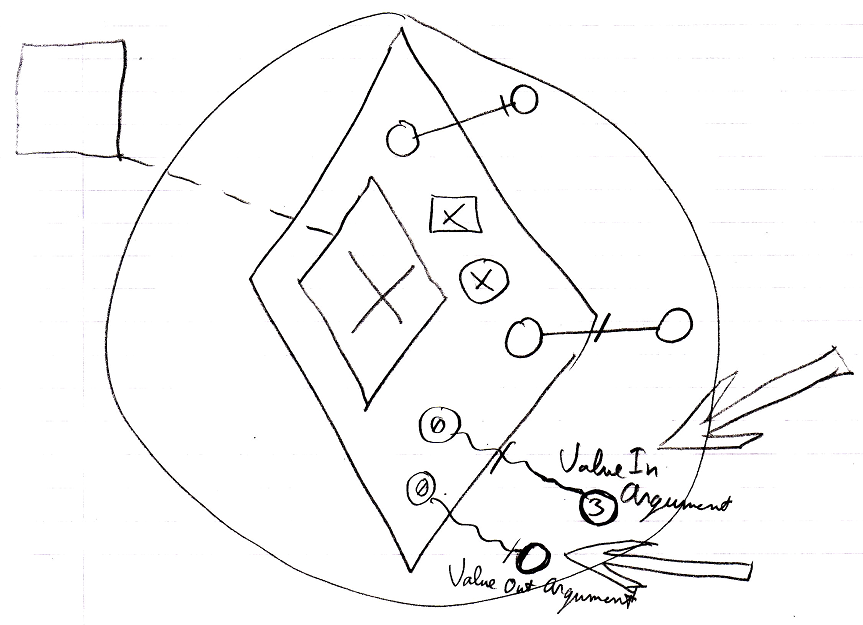
- Private contents are *not* created  
(private objects, clauses and command calls)



- Reference and Object Out parameters are already assigned

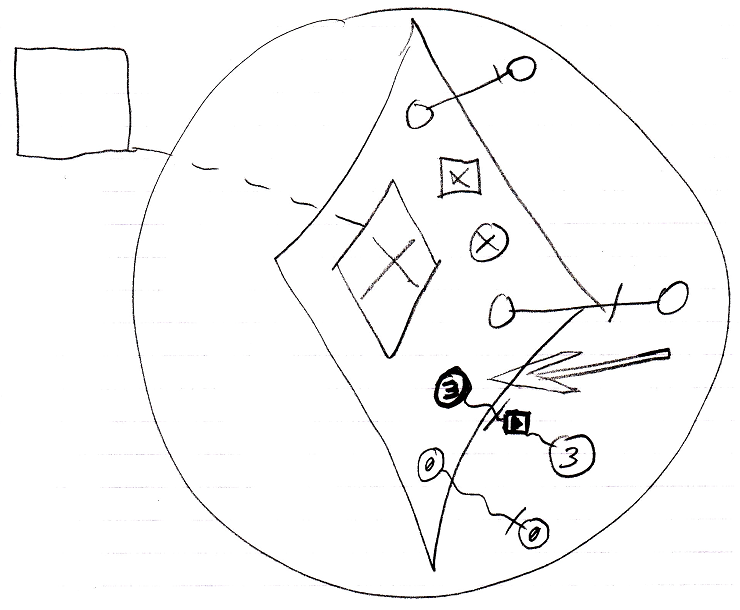


- Value parameter assignments refer to their sources and targets

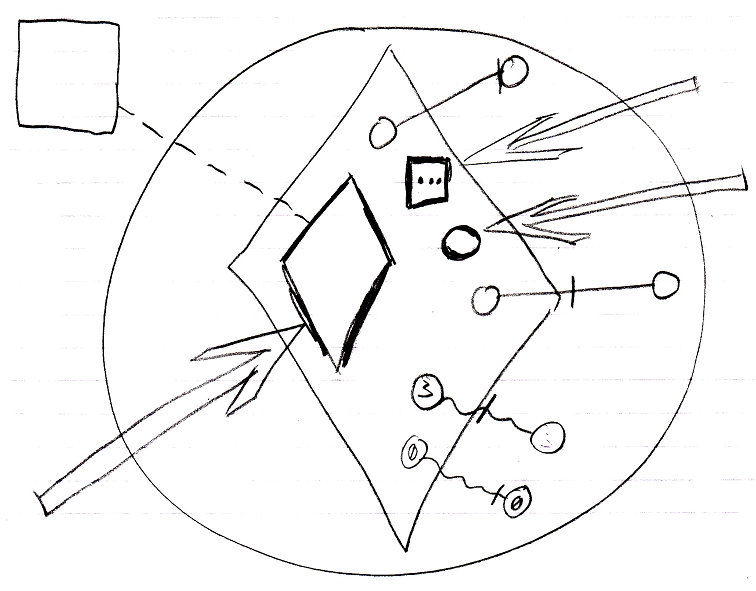


Right before a command executes:

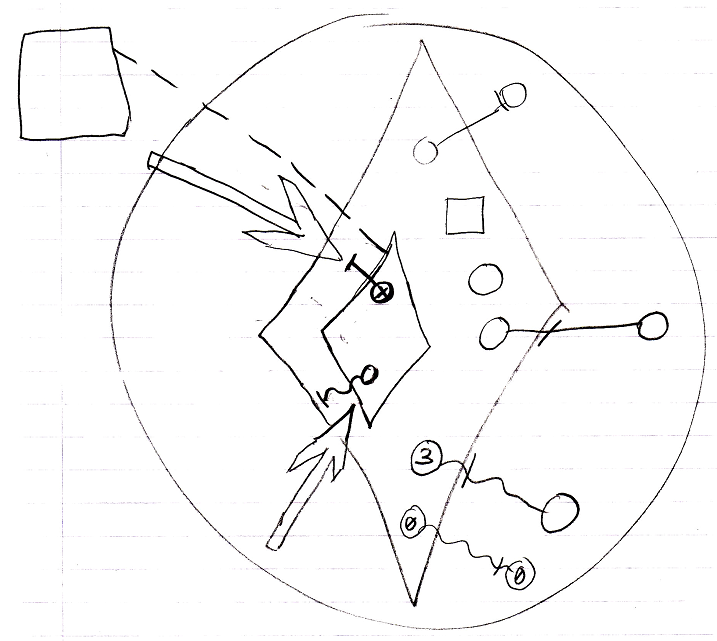
- Value In parameter assignments are executed



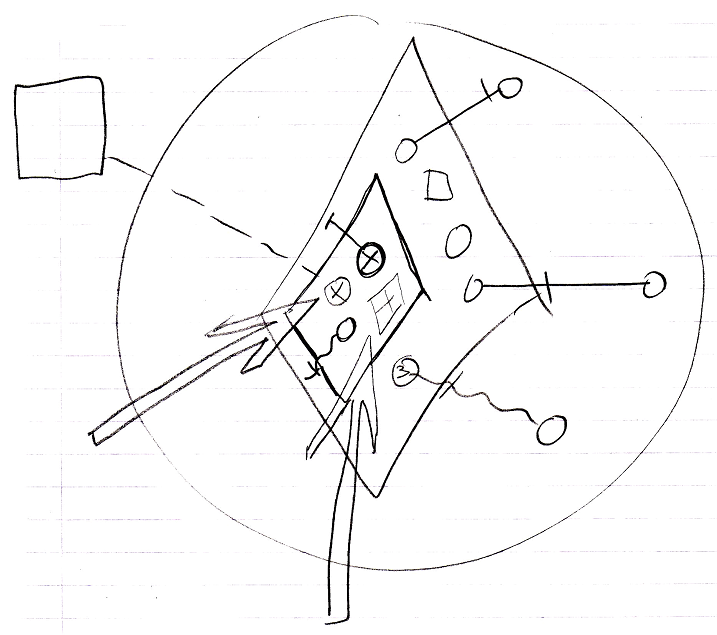
- Private contents are created  
(private objects, clauses and command calls)  
(copied from the command’s definition)  
(The definition of the parent command is not shown in the diagram.)



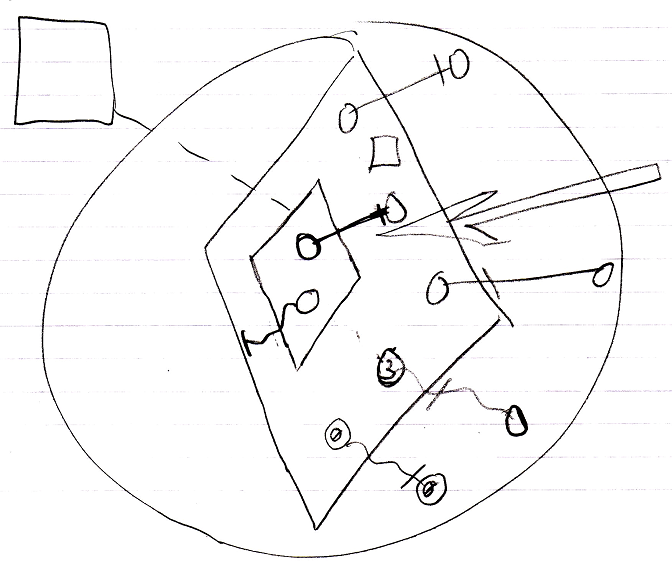
- *Sub*-commands’ parameters / public contents are created  
(copied from the sub-command’s definition)  
(The contents of the definition of the sub-command are not shown in the diagram.)



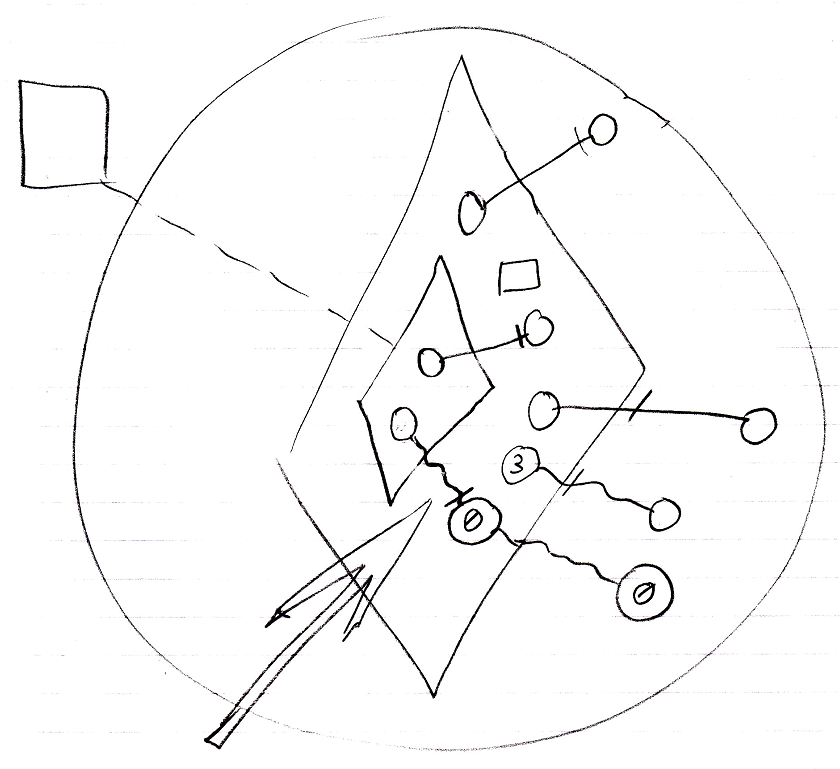
- Mind, that the sub-command’s private contents are not created, which includes its private objects, command calls and clauses.



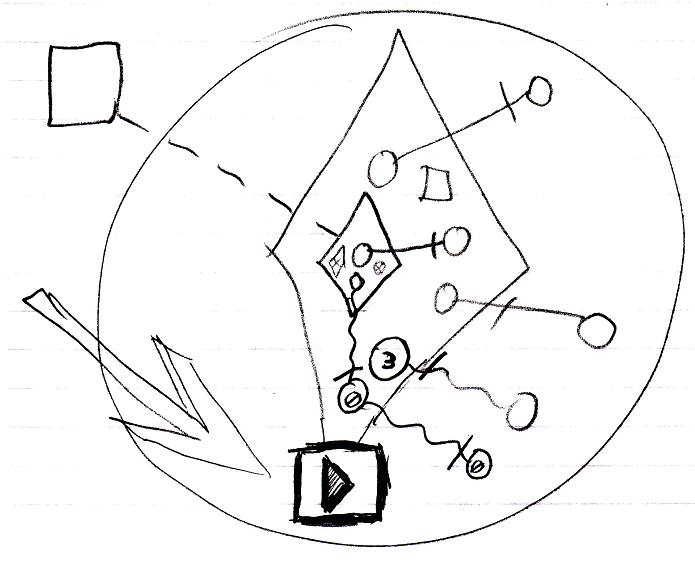
- *Sub*-commands’ Reference and Object Out parameters are assigned  
(targets copied from the parent command’s definition)  
(The definition of the parent command is not shown in the diagram.)



- Sub-commands’ Value parameter assignments refer to their sources and targets  
(copied from the parent command’s definition)

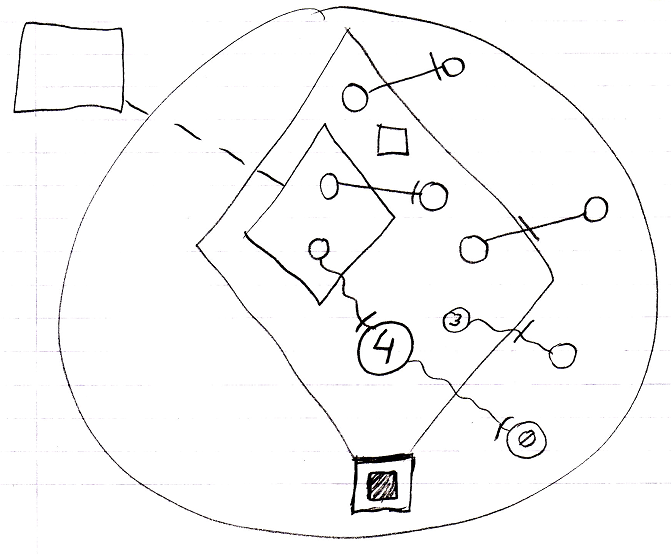


Then the command runs:

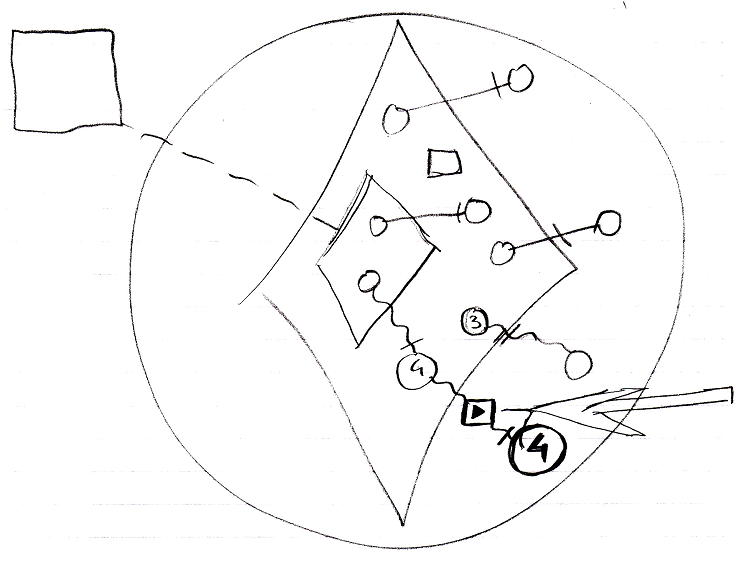


which means it runs all its sub-commands one by one, following the same procedure. The sub-command’s public content needed to be created in order to instantiate the parameter passings, part of the parent command.

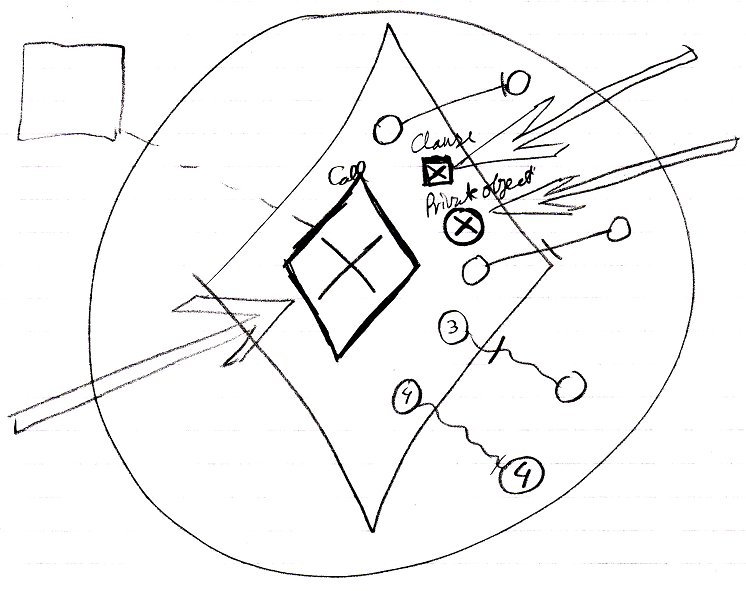
After a command has finished:



- Value Out parameter assignments are executed



- Private contents are released  
(private objects, clauses and command calls)



### Creation Behavior of Clauses

The section *Creation Behavior of Calls* talked about delaying the creation of a call’s private contents, until the command is about to be run, while the public contents of a command call are there straight away, as soon as the command call is created.

But if a command object does not have a definition, then it defines its own definition. For command objects that define their own definition, private contents *are* created all the time, because nothing else defines its private contents but the object itself.

This also counts for clauses.

Clauses are like command definitions inside another command.

Even when a clause is an execution, it is also a command definition. Command definitions are created permanently, so clauses are created permanently too, as well as clauses inside other clauses. Active clauses have added behavior compared to other sub-commands (command calls). They are like command definitions inside another command. An active clause’s private data is already created. Even when the clause structure inside a command is very deep, the *whole* depth of the clause structure is recursively created when the parent command is created. The clause structure can not have circularities and is always a limited tree structure, so that the process of creating the whole clause structure can never hang or anything like that.

Other active sub-commands (for instance command calls) behave differently. A command calls’ *private* data is not created until the command is actually run.

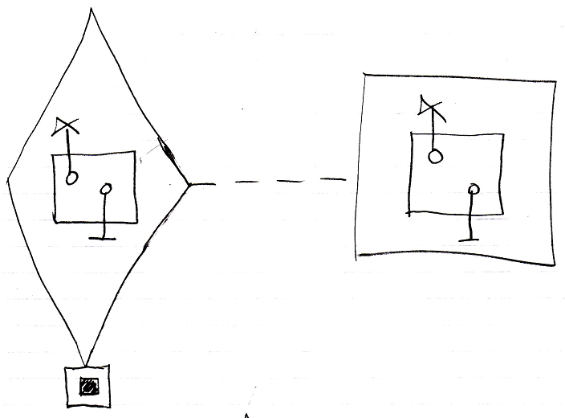
Clauses being permanently created as long as the parent command is created even counts for clauses inside a command call. Right before a call is executed, its private contents are created, including the whole depth of its clauses. In theory the definition of the clause could be pointing to the clauses inside the command call’s definition. The private contents of the clause could be created only just before the clause is run. But this is not done. As soon as a clause in a command call is copied from the definition, the clause has no connection anymore to the clause in the definition. Therefore, it needs to define its own private contents.

Note, that though everything of the clauses is created, parameters of an active clause are only *assigned* right before the clause is run.

#### In a Diagram

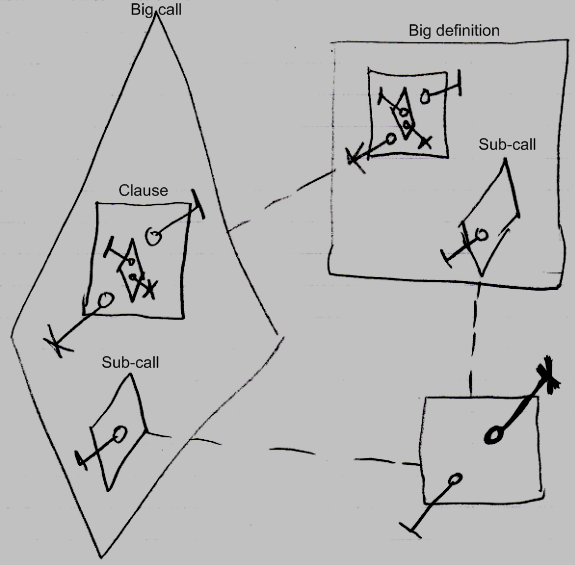
Just like other commands, that do not have a definition, a clause’s contents are created all the time, including its private contents.

Clauses being permanently created as long as the parent command is created even counts for clauses inside a command call. Right before a call is executed, its private contents are created, including the whole depth of its clauses.



In theory the definition of the clause could be pointing to the clause inside the command call’s definition. The private contents of the clause could be created only just before the clause is run. But this is not done. As soon as a clause in a command call is copied from the definition, the clause has no connection anymore to the clause in the definition. Therefore, it needs to define its own private contents.

Even when the clause structure inside a command is very deep, the *whole* depth of the clause structure is recursively created when the parent command is created.



The sub-call inside the big call and inside the big definition only have the public parameter created, not their private contents as neither of them is running. But the clause, with all its contents, are created inside the big call all anyway.

### Creation Behavior of ‘Inactive Calls’

Command calls inside a parent command have special creation behavior: their publics are created as soon as the parent command is created, but their privates are only created when the command call is about to be run.

There is also the *inactive* form of a command call inside a parent command.

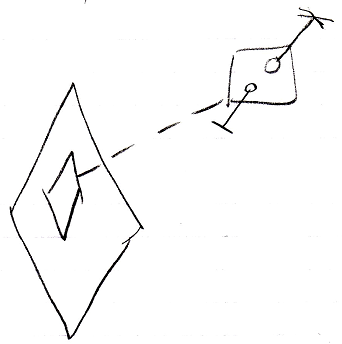
Special creation behavior does *not* count for inactive command objects inside a parent command, that have a class redirection to a command definition. This looks like the inactive form of a command call, but this kind of object does not have special creation behavior like that. It is an uncommon situation. But an inactive command inside a parent command with a class reference to a definition *can* be referenced, unlike its active form. Therefore, it can also be *class* referenced by a call. In that case its privates and publics had better be there more permantly, or the *call* to it can not instantiate private contents at all.

Creating its private contents, does not create a recursive creation or anything: the inactive call’s own private calls do not create *their* private contents, so there is no recursion there.

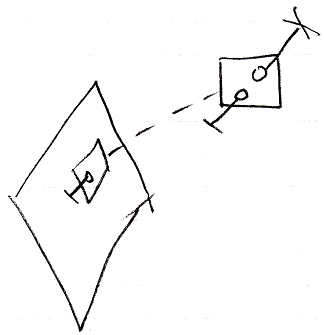
#### In a Diagram

The article *Creation Behavior of ‘Inactive Calls’* already explained this behavior conceptually. The current article further clarifies the idea using diagrams.

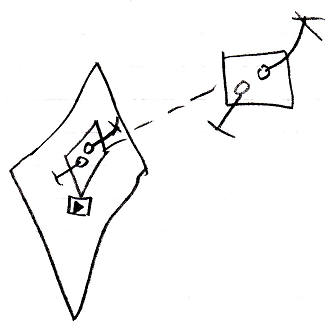
Command calls inside a parent command have special creation behavior:



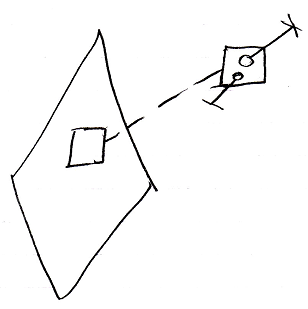
Their publics are created as soon as the parent command is created



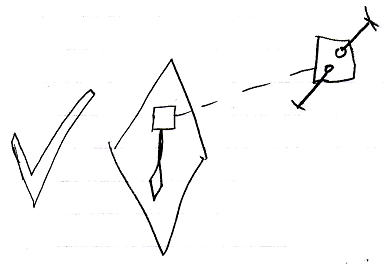
but their privates are only created when the command call is about to be run.



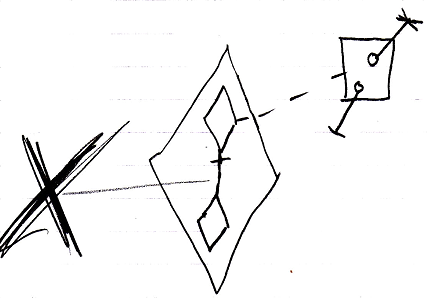
Special creation behavior does *not* count for inactive command objects inside a parent command, that have a class redirection to a command definition.



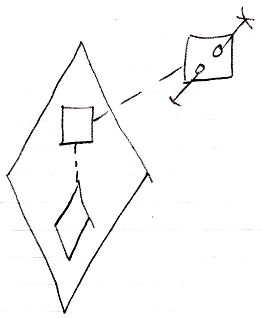
This looks like the inactive form of a command call, but this kind of object does not have special creation behavior like that. It is an uncommon situation. But an inactive command inside a parent command with a class reference to a definition *can* be referenced



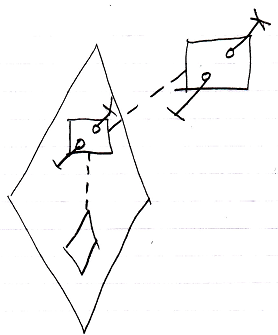
Unlike its active form, which can not be referenced.



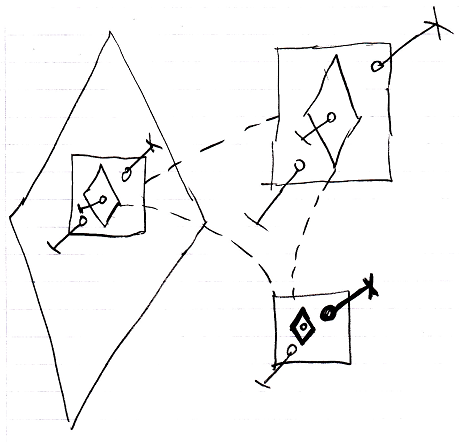
Therefore, it can also be *class* referenced by a call.



In that case its privates and publics had better be there more permantly, or the *call* to it can not instantiate private contents at all.



Creating its private contents, does not create a recursive creation or anything: the inactive call’s own private calls do not create *their* private contents, so there is no recursion there.



### No Overhead of Command Creation

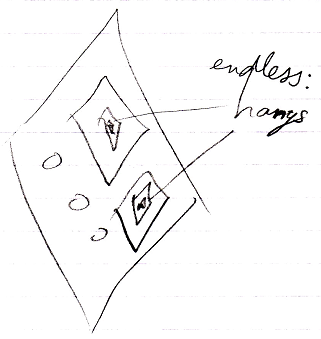
One of the reasons why private contents of a call are only created just before the call is run, is because this prevents overhead of creation. If you would have to create the whole call structure when the great grandparent of commands is created, then this is not only a lot of work in one blow, but also, a lot of command call objects are created, that never get to run in the first place, because it only runs under certain conditions. Delayed creation of the private contents of a command call prevents this overhead of procedure creation and creates a neat and steady rate of procedure creation.

### No Circular Command Creation

Another reason why private contents of a call are only created just before the call is run, is because this prevents circular creation of commands. Some command may call another command and that command may call the first one again. Command calls are usually private, so if you would create all possible command calls, you end up creating an endless recurrence of command creations, while in reality, the recurrence will be broken by some conditional execution of one of the command calls. Creating private contents of command calls prevents this circular creation and only creates a command object when it will actually run.

#### In a Diagram

The concept of No Circular Command Creation has already been explained in the article *No Circular Command Creation*. The current article shows what circular creation could look like



### No Private Contents in a Call in a Definition

A definition is always dormant, and never runs. So also the *calls* inside a definition will never run. Therefore, the private contents of calls inside a definition are *never* created. A call in a definition never shows the call’s private contents. The call at most shows its parameters, so the public contents of the command call. Only the *definition* of the called command will show private contents. So you have to hop to the definition of a call to see the private contents of the command.

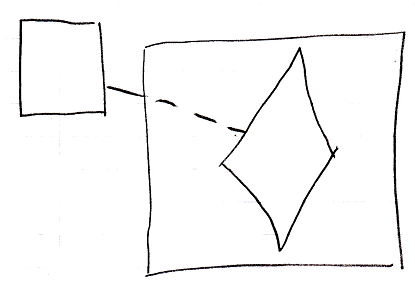
Not creating a call’s private contents before it even runs, takes away discussion about when to display and when not to display a command’s private contents.

Bear in mind, that when an executable object does not redirect its definition, it has to define its own private contents, because nothing else defines its private contents but he himself. But *calls inside* such a definition, only have their *public* contents are created again.

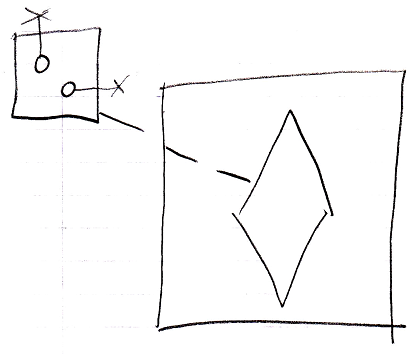
#### In a Diagram

This section repeats the story, but now demonstrates the concept using diagrams.

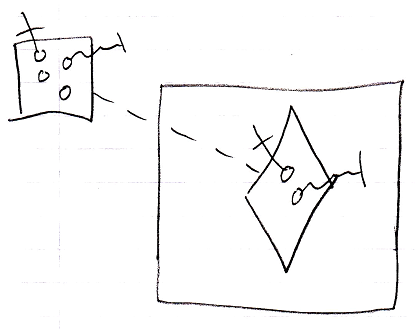
A definition is always dormant, and never runs. So also the *calls* inside a definition will never run.



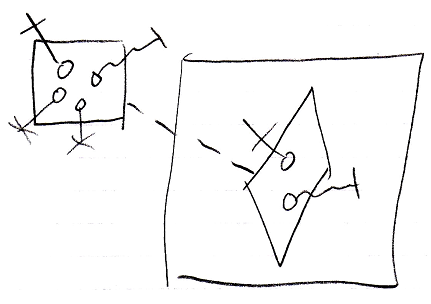
Therefore, the private contents of calls inside a definition are *never* created.



A call in a definition never shows the call’s private contents. The call at most shows its parameters, so the public contents of the command call.



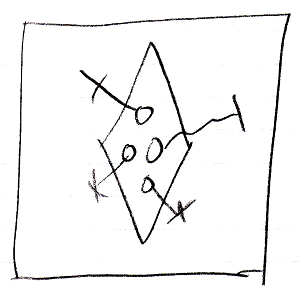
Only the *definition* of the called command will show private contents.



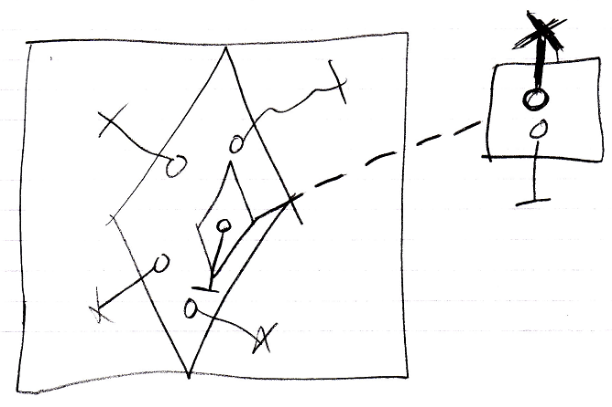
So you have to hop to the definition of a call to see the private contents of the command.

Not creating a call’s private contents before it even runs, takes away discussion about when to display and when not to display a command’s private contents.

Bear in mind, that when an executable object does not redirect its definition, it has to define its own private contents, because nothing else defines its private contents but he himself.



But *calls inside* such a definition, only have their *public* contents are created again.



### A Call in a Call Shows Privates When Running

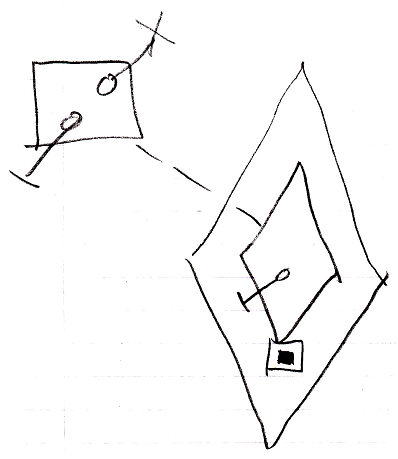
A call inside another call only contains its private content when it is actually running. If a call is not running, then you would have to hop to the definition of a call to see the private contents of the command.

Not creating a call’s private contents before it even runs, takes away discussion about when to display and when not to display a command’s private contents.

#### In a Diagram

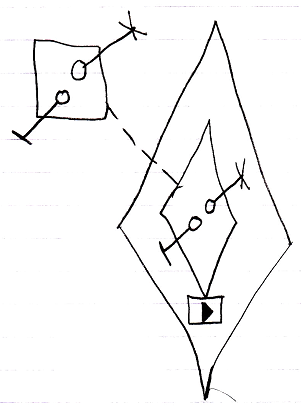
A call inside another call only contains its private content when it is actually running.

If a call is not running:



Then you would have to hop to the definition of a call to see the private contents of the command.

When a call is running, you do see its private contents:



Not creating a call’s private contents before it even runs, takes away discussion about when to display and when not to display a command’s private contents.

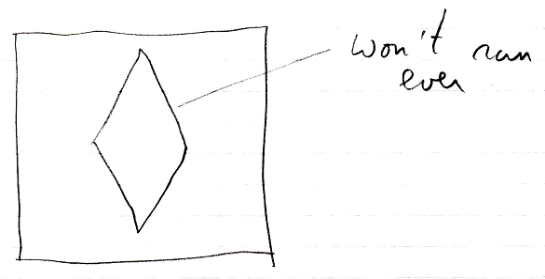
### Active Command in Inactive Command

A clause in a clause only ever runs when its top parent command is an active command object. If the top parent command is an inactive command object, for instance a command definition, then even an *active* clause in it is dormant.

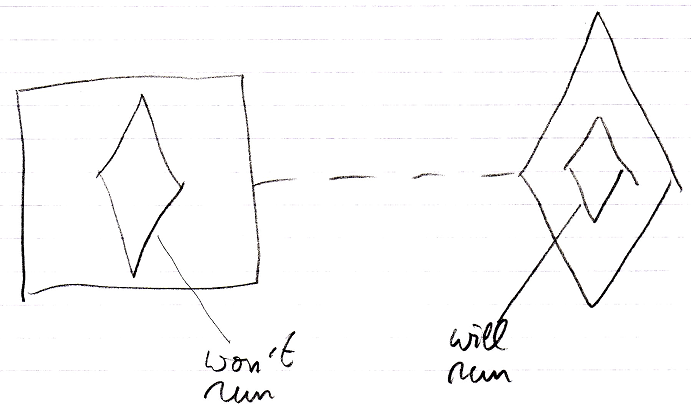
An executable command inside a command definition can not be run, because its parent is dormant, and an executable sub-command can not be referenced from elsewhere either.

#### In a Diagram

A clause in a clause only ever runs when its top parent command is an active command object. If the top parent command is an inactive command object, for instance a command definition, then even an *active* clause in it is dormant.



The clause will only ever run in a *call* to the definition, but then it is a copy of the definition’s clause:



An executable command inside a command definition can not be run, because its parent is dormant, and an executable sub-command can not be referenced from elsewhere either.