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

## Ideas

Not showing private members or the contents of procedures is an important subject for the display of the diagrams.

The definition of something can just as well be shown inside a reference to the definition. It’s a setting whether you’ll show the full definition in a reference or just the public things. It is best if you can’t *change* the definition from a reference to a definition.

The definition will be shown next to the diagram anyway. You can change it there. That will work more intuitive.

CL,

In concepts zetten:

Precieze tijd-statistieken van alle processtappen,

zodat je precies weet in welk stukje van het proces

de bottleneck zit, niet alleen gewoon dat het proces

lang duurt. Dan weet je ook waar het aan ligt,

anders is het gissen.

JJ

Command Basics,

2008-08-17

It seems, that when you have active command references

inside different active parent commands, you get a problem.

Multiple parent commands need to wait on the referenced

command object to finish.

JJ

Command Basics,

The following Basic Command articles do not have a diagram expression article:

- Execute Once

- Resolution When Not Allowed For Commands

- Parent Controls Its Sub-Executions

- Sub-Commands Not Manually Started

- No Overhead of Command Creation

- No Circular Command Creation

JJ

Start & Stop ,

2008-08-29

Vinkje is ook een mooi symbool voor voltooid.

JJ

Command Basics,

2008-08-31

For instance: the rule ‘sub-commands are never referenced’, may be changed to being able to reference sub-commands after all, but never to be able to execute a sub-command through a reference, even if the reference is active. But this change of rules is just an example. It might prove not to be practicle after all.

> Perhaps change that rule for real.

JJ

Command,

2008-09-16

I've seen others call the following execution commands:

<< reverse

>> fast foreward

JJ

Command,

2008-09-16

I've seen others use the following names for the following execution commands:

|< skip backward

>| skip foreward.

>> foreward

<< rewind

JJ

CL,

2008-09-18

Execution data.

Transport time

Ways:

Network (133.1.9.23) - Memory - Disk (\\local\E:)

14 sec

Disk - Memory

15 sec

Processing time

4 sec

<other times>

JJ

It’s *procedures* that are called. When you speak of calling an *object*, you’re talking about calling a procedure of an object (or a procedure of one of its sub objects).

Commands,

2004

There’s a method for turning a procedure into a class.

Say a procedure has input objects and output objects. When you call the procedure you set the input objects. After the call you can read the output objects.

To create a class out of the procedure you just take over all of the procedure’s objects. The procedure itself, stripped from objects, becomes a member of the type too.

You can freely read and write the objects. However, the procedure must be occasionally run.

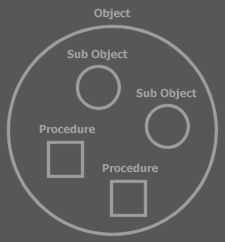
The benefits from making a type out of a procedure is that you can dynamically write and write input objects and read output objects, without having to run a procedure all the time.

JJ

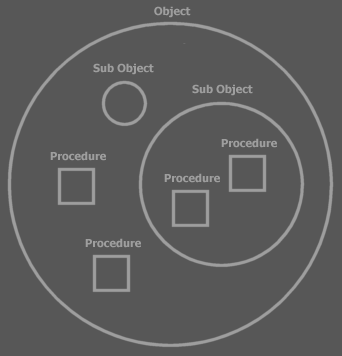
# From the original Symbol documentation

## Introduction, Procedures

Apart from a collection of sub objects, an object contains *procedures*.



Sub objects again contain procedures.



Software can’t execute without procedures. When you run a procedure, an object *does something*.

A button could have a **Set Text** procedure for instance, which sets the text displayed on the button.

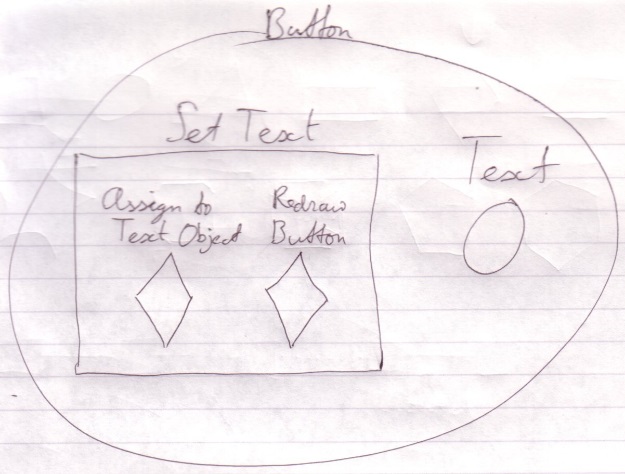


Procedures are denoted by *squares*.

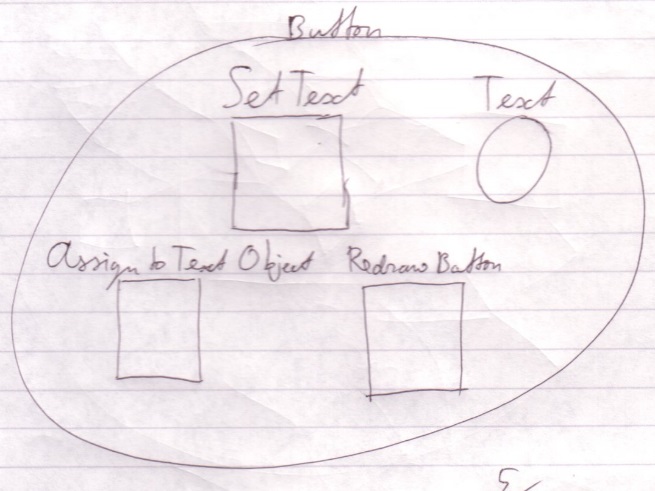
Other objects could also have a **Set Text** procedure, therefore to identify the **Set Text** procedure of the ***Button*** object you notate:

Button . Set Text

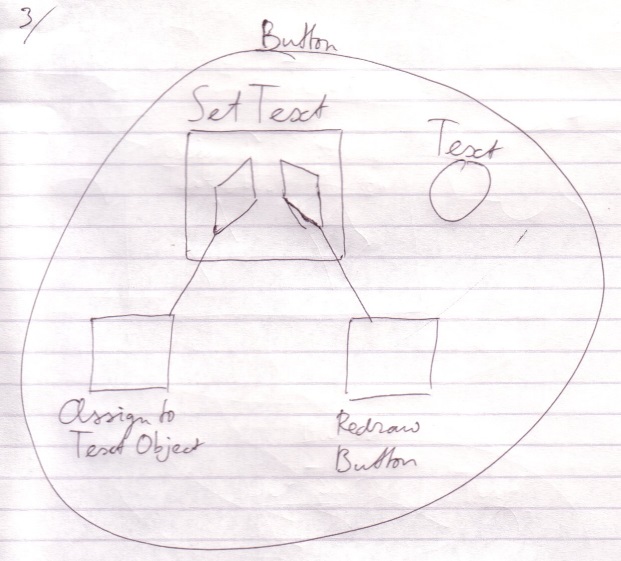
A procedure itself consists of sub procedures: the separate steps of the procedures.



Sub procedures are usually not embedded inside other procedures. They are defined separately:

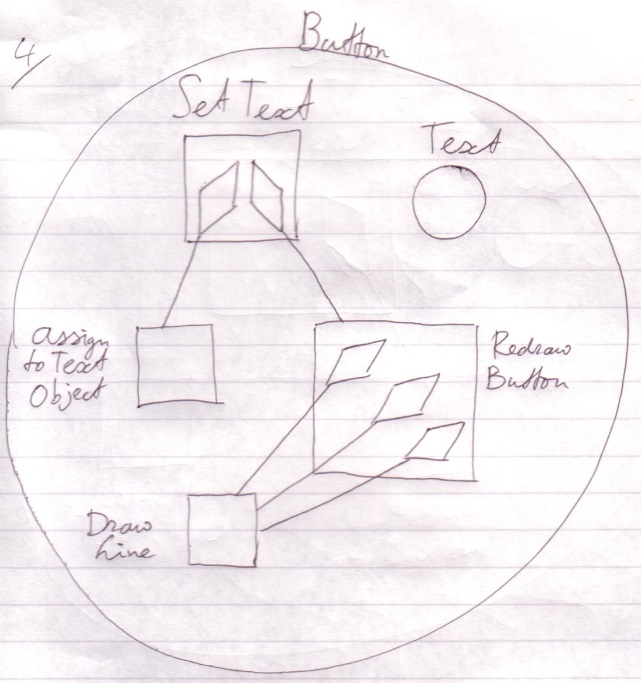


And the sub procedures are *called* from the super procedure:



Calling a procedure has almost the same effect as inserting the called procedure there where you call it.

The **Redraw Button** procedure is rather complex. In this procedure lines of the button are drawn, the text is drawn and whatever else makes up the display of the button. The steps of this redrawing are delegated by yet again calling other procedures. **Redraw Button** could for instance call the **Draw Line** procedure a number of times. So sub procedures of the **Set Text** procedure are themselves composed of sub procedures.

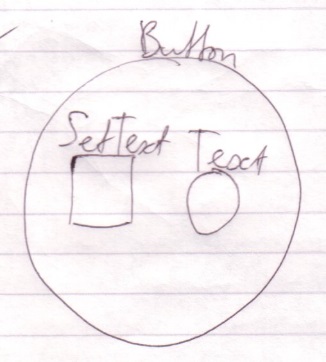


You can also see here that separately defining a procedure can lead to the *reuse* of procedures. The **Draw Line** procedure is reused three times in the example above. There are many procedures already defined, that you will use one way or another. Most of these procedures are encapsulated in a type. There’s for instance a **File** type that contains procedures with which to control a single computer file.

So where does it end? Procedures would continue to delegate to one another and nothing would really actually happen. Well, it ends at a special group of procedures that don’t call other procedures anymore. Each of those procedures executes a so called machine instruction: a basic instruction that is sent to the computer’s central processing unit (CPU) and make the *computer* *do* something. The CPU performs a hardware defined *machine* procedure.

That way there can develop a big procedure call tree-out, which makes a single procedure consist of many, many machine instructions, ranging from tens to thousands of machine instructions or even more.

Justs like with the kitchen example: you don’t necessarily need to see the details of the procedure in order to understand what the effect **Button . Set Text** is.



Procedures can also be called *operations*, *functions* or *routines* and there are even more synonyms. But I will usually stick to the term *procedure*.

## Procedure Basics

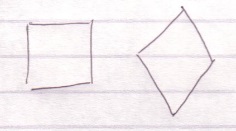
< a diamond is also notated pointier than a square. So this is be a diamond too:

[Picture with a pointy square]

>

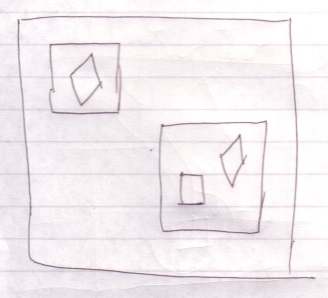
##### Notation Elements

*Squares* and *diamonds* are procedure symbols. They represent elements of the procedure structure.

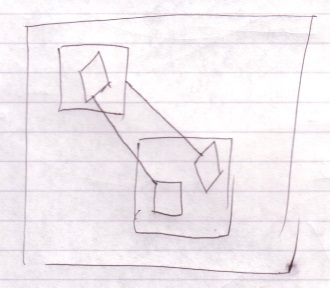


The square is the usual symbol for a procedure. The diamond is a special symbol, as I will show lateron.

Relations between procedures are expressed by *containment*:



And by connecting them with lines, or *lines:*



There can only be solid or dotted lines between procedure symbols.

##### Procedures

Procedures are denoted with squares:



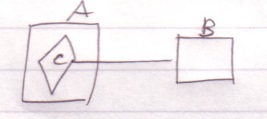
To make two squares be the same procedure you connect them with a line:



A call to either square is a call to the same procedure.

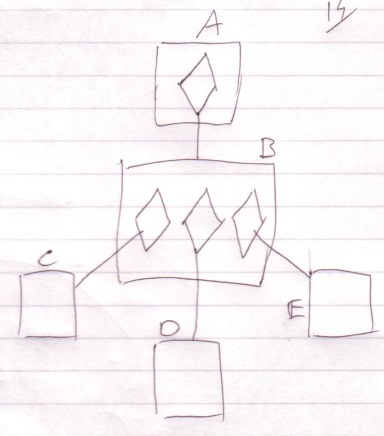
##### Procedure Call

To make one procedure call another procedure, you put a diamond in the calling procedure and you connect it to the procedure to call.



You can say **A** calls **B**, or **B** is called from **A**. You could say that the line is directed outwards: the call line first exists a square, to next find its target procedure. The connected symbols here represent the same procedure. The side effect of a diamond though, is that it *executes*. **C** is now part of procedure **A** and it will execute when **A** executes. It’s like the code of **B** is inserted right into procedure **A**. Considering that, it seems that the direction of the line doesn’t really matter: both symbols simply represent the same procedure. But a procedure does have a definition in one place while the other symbols are calls or references to it. The topic of direction is looked at closer in the *Lines* chapter, but I’ll tell you now that the direction of a line is usually outwards.

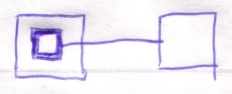
If one square is called, its contained squares are called too.



When **A** is called, **B** is called. When **B** is called, **C**, **D** and **E** are called.

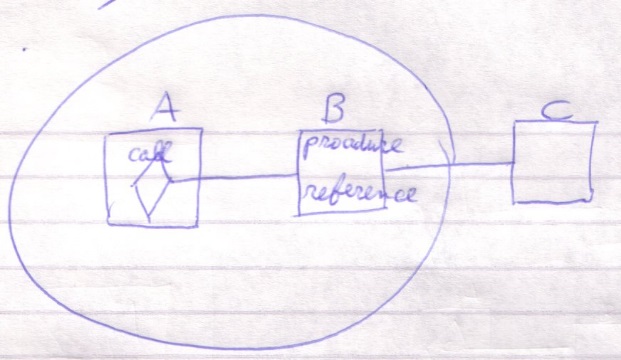
##### Procedure Reference

A diamond connected to another procedure is a call. A square connected to another procedure is a mere *reference* to the procedure.



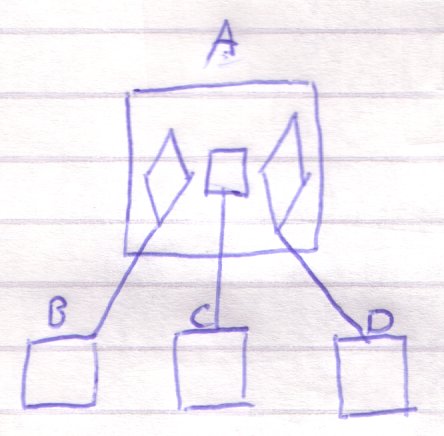
A call to any square that represents the same procedure is a call to the same procedure. A reference to either of the procedure references is a reference to the same procedure.

Therefore, when you call a procedure reference, the referenced procedure is called:



When **A** calls **B**, it’s actually calling **C**.

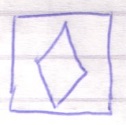
When a procedure is called, the contained procedure *references* are *not* called.



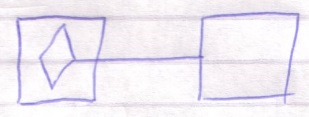
When **A** is called, **B** and **D** are called, but not **C**.

##### Procedure Clause

A diamond without a line:



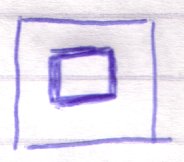
Also executes, when its parent procedure executes, but it doesn’t delegate to another procedure, like a call does:



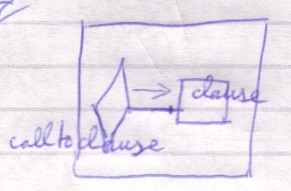
It is an undelegated part of a procedure, called a *clause*.

Because a diamond doesn’t have to be a call, it is also called an *execution* or an *execution point*.

An undelegated square inside a procedure is also considered a clause.



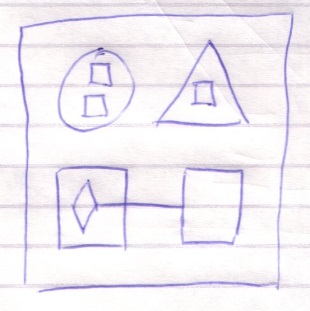
but it doesn’t execute, unless you call it.



A splineless diamond is also called an *active clause*, while a splineless square is also called an *inactive clause*.

##### Procedure Contents

A procedure can contain anything an object can. Anything you can declare inside a type can also be declared in a procedure.



That way procedures are like types, but procedures have special characteristics that I’ll lay out later.

The objects inside a procedure can be regarded:

* Input parameters
* Output parameters
* Throughput parameters
* Return values
* Local variables

The input parameters of a procedure are its writable objects. The output values are its readable objects. Objects that are read-write are like in-out or thru parameters. One readable object can be chosen to be the return value, which promotes it to being the main output parameter. This does not give it extra capabilities, only an extra notation in certain places. The return value is denoted in a diagram by putting the term **Return** near one of the parameters. The private objects are the procedure’s local variables. If **A** is an input parameter and **B** is an output parameter, you can also say that the procedure *takes* **A** and *gives* **B**.

Squares inside procedures are non executing clauses or procedure references. Diamonds inside procedures are executing clauses or procedure calls.

##### Procedure Versus Type

*Procedures* differ from *types* in the following ways:

* They can execute
* A new procedure object is created *on each call* to the procedure and the procedure object is destroyed soon after the call.
* The public writable objects are written only right before the call.
* The public readable objects are read only right after the call.

##### Required and Optional Parameters

As a procedure is like a type it may seem that you have full freedom regarding which objects you write and which you don’t. However, many parameters of procedures are *required* parameters. This means that you *must* write something to it before the call. Optional parameters are ones that do not necessarily need to be written before the call. Required parameters are there to on one hand ensure a parameter holds a right value. That however, could have been done by the procedure’s initializing the value itself. The main point of required parameters is that a lot of times the function of a procedure just doesn’t make sense unless you write the parameters. The programmer is made extra aware of that by making the parameter required.

In diagram code, optional parameters are denoted by displaying the word **Optional** near the optional parameter. The other parameters are required.

## Executions & Parameters

This section discusses all issues around executing procedures and using parameters.

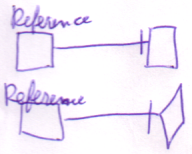
### Terms

Here I introduce basic terms in the area of executions and parameters. Here’s a quick reference list.

* Reference
* Call
* Definition
* Clause
* Active
* Inactive
* Active clause
* Inactive clause
* Sub clause
* Sub procedure
* Parameter
* Argument
* Executor
* Execution

###### Reference

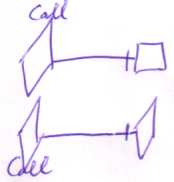
The following are procedure references:



It’s a square that is redirected. It’s always a *square* with a *reference line*. It is a procedure that redirects, but *does not* execute. It can point either to a square or a diamond.

###### Call

The following are calls:



It’s a diamond that is redirected. It’s always a *diamond* with a *reference line*. It executes a procedure defined elsewhere. Therefore it is a call. It can point either to a square or a diamond.

###### Definition

A definition is a procedure symbol with no reference line at all.



The reference line should not exist at all, not just not be shown. It’s a definition, because the contents of the procedure aren’t defined elsewhere, but defined in the symbol itself.

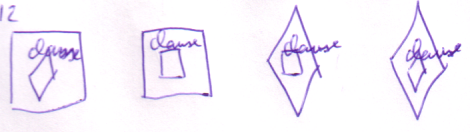
###### Clause

A procedure symbol inside another procedure symbol:



without a reference line is called a clause.

A clause is a definition contained in a procedure:



This type of definition is usually not called a definition. The clause must be directly contained by a procedure. So this is not a clause:



The clause can be a diamond or a square. Its container can also be a diamond as well as a square. A clause does not have a reference line, or it would have been a reference or a call, not a clause.

###### Active and Inactive

A diamond executes, a square does not. A diamond is said to be active, while a square is inactive.

###### Active Clause

A clause that is a diamond, is called an active clause:

|  |  |
| --- | --- |
|  |  |

It’s called active, because it executes.

###### Inactive Clause

An inactive clause is a clause that doesn’t execute. So it’s a clause that is a square:

|  |  |
| --- | --- |
|  |  |

It won’t execute unless you call it.

###### Sub Clause

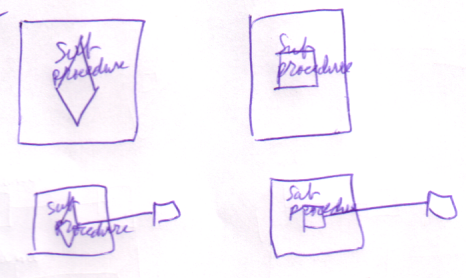
A sub clause is a clause inside a clause. So it’s a procedure symbol without a reference line in a procedure symbol without a reference line in a procedure symbol, for instance:



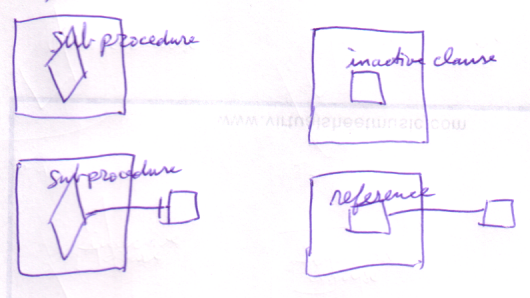
It doesn’t matter whether any of the symbols is a square or a diamond.

###### Sub Procedure

A sub procedure is any procedure symbol inside another procedure symbol.



But usually it’s only the diamonds that are referred to as sub procedures. That’s because the execution of a procedure is made up of the execution of it’s contained diamonds combined.



###### Executor and Execution

Any diamond symbol is called an executor, because it can execute. A single diamond can execute multiple times. An individual execution of a diamond is called an *execution*.

It’s allowed to call an executor an execution, but usually only while it’s executing.

### Execution Basics

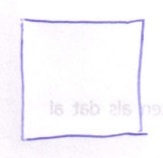
<Some things here may have a place in Execution Order>

<Which situations can only reside in a procedure?>

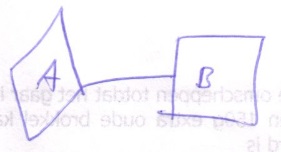
This section covers some basic execution situations. Each situation has certain implications. Usually the implications are quite <voor de hand liggend>, but noteworthy nevertheless.

<Deze inleiding is te denigrerend over de sectie>

###### Squares and Diamonds



A square doesn’t execute. Well, they only execute when you call it.

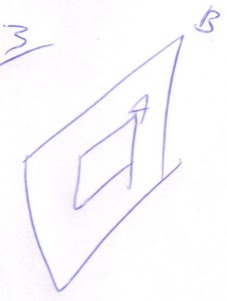


A is a call to B.

But then again: A is the one that’s executing, not B. So diamonds execute, and squares do not.

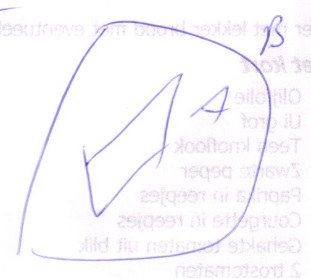
###### Diamonds Execute When Direct Parent Executes

A diamond doesn’t execute automatically. It only executes when the parent symbol executes.

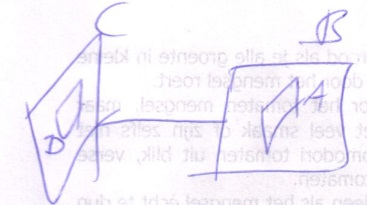


A executes only as B executes.

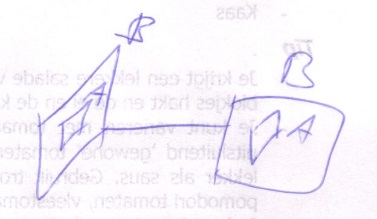
So diamonds inside squares don’t really execute.



But if you call B, then you get the following:



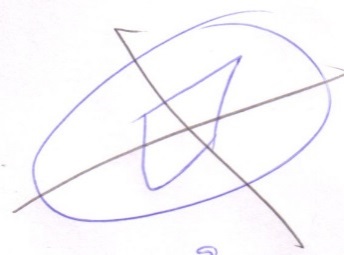
Actually, The letters distinguish the different symbols. If you only use letters to distinguish which procedure definition it’s about, you get the following:



So in that sense, diamond A does execute. But not the definition executes, only the call executes.

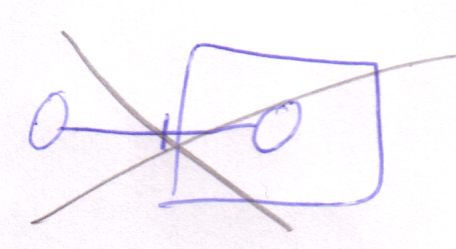
###### Diamonds Can only be Directly Inside a Procedure Symbol, Diamonds Can’t be Directly Inside an Object Symbol

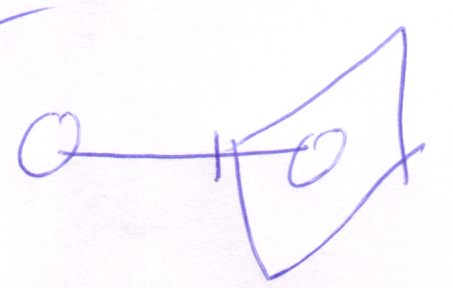
Because execution can only take place in a procedure, a diamond can’t be directly inside an object Symbol



###### You Can Only Access Members of a Diamond

You can (usually) only access members of a diamond, because a square is never created, just like you can’t access members of a non created object.





###### You can Only Access Members of a Diamond, while the Diamond is executing

Because a diamond is only created when it’s executing, you can only access members when the diamond is in execution.

Some members of a procedure can be static. In that case the member belongs to the procedure definition. Those member are the same for any reference or call to a procedure. Static member of a procedure can be freely accessed through any square or diamond that represents it.

###### The Diamond Executor Can Access Members Only Just before and Just After the Execution

If the diamond executor accesses something of a diamond, it’ll only access it just before and just after the call.

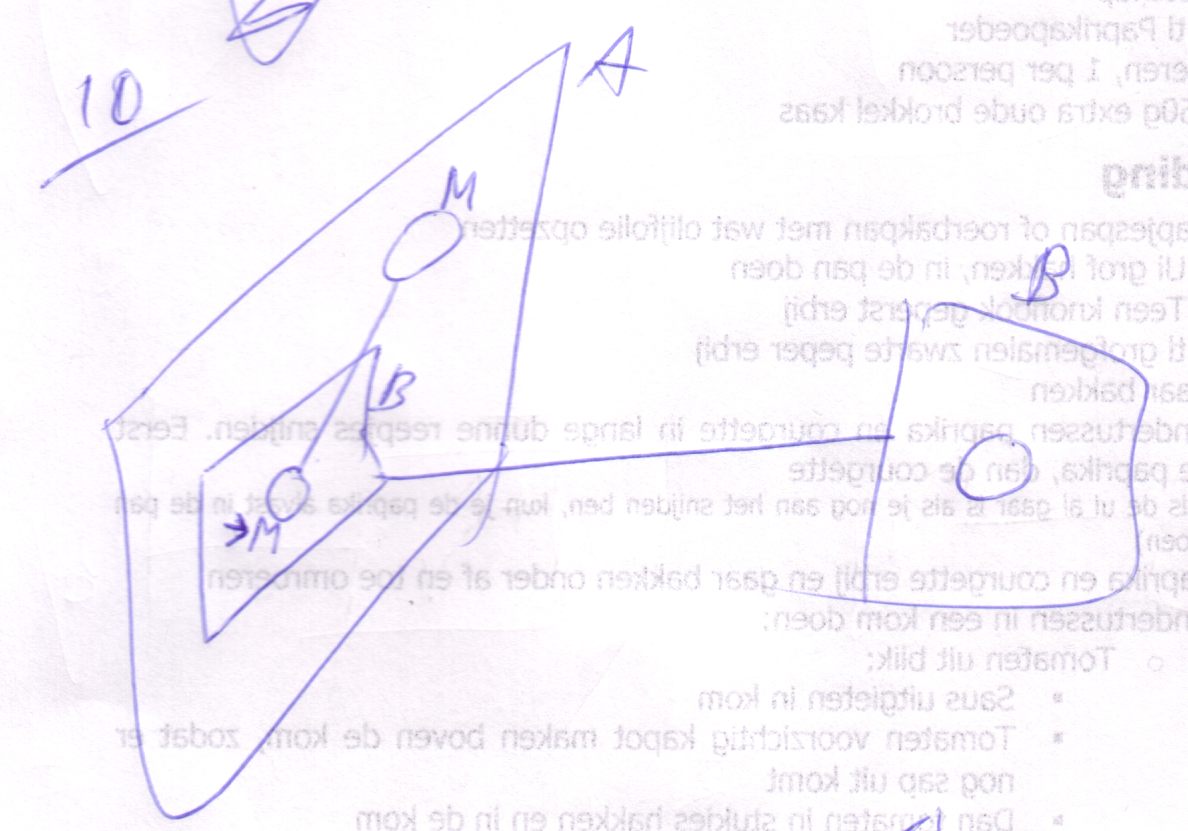
<Don’t know the notation, really>

In fact the diamond executor will only write members just before the call and will only read members just after the call.

<Don’t forget that you don’t need to read or write necessarily, you can also just call a member of the parameter instead of read or write it… explore that>

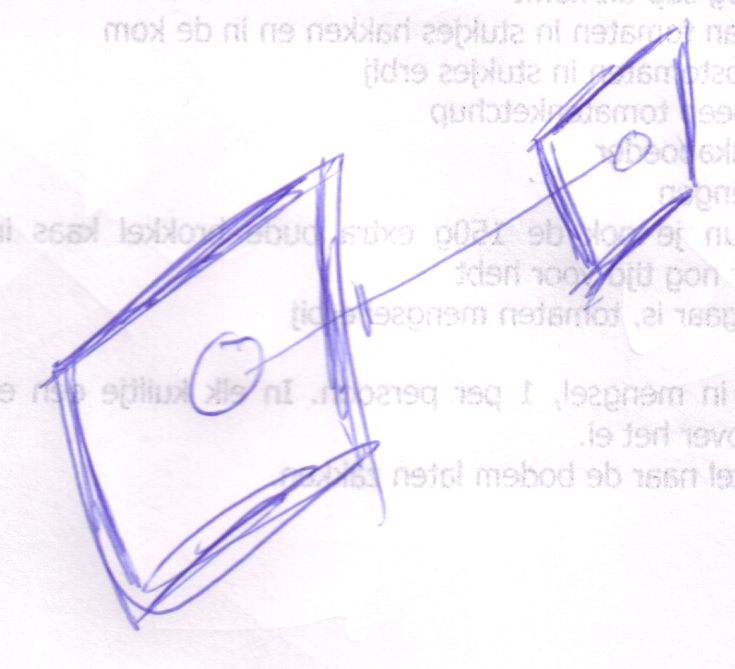
###### Accessing a Diamond Member During a Call

So how will you access a diamond member *during* a call? Well, usually only procedures called by the executing diamond can access the call parent.



Diamond A executes, then diamond B executes and accesses a member of diamond A. It’s not the procedure definition, the square, that accesses a member of diamond A, but it’s the specific call to the procedure that access diamond A. So usually you’ll only see child diamonds accessing parent diamonds, on top of the more common situation of write access before a call and read access after.

Multi-threading, which is …, can also make you able to access a diamond in execution, but this is usually not a good idea. Restrictions should be opposed to multi-threading to avoid such situations.



Two sibling procedures are executing at the same time, in different threads, which makes it possible for one executing procedure to access the other. Thread control makes it possible to avoid such volatile situations.

(the fact that the diamonds are drawn with thicker lines above, says that they are both in execution. If they wouldn’t be thicker drawn, then the diagram above doesn’t explicitly show that there is a multithread situation. Regarding it single threadedly, the situation above is just the bigger diamond executing first, after which the smaller diamond uses its return value.)

< What happens when two threads try to initiate the same diamond or a thread tries to initiate a diamond that’s already executing. I want that one figured out>

<consider the example of diamond reference to diamond in the main argument reference example >

###### Passing an object reference to a procedure



When you pass an object reference to a procedure, the procedure can access the object.

In the situation above it seems the procedure could access the object anyway (because a procedure can access everything accessible to the object that contains it). It *can*, but in this case the *caller* decides which object to point to, not the called.

###### Procedure can set object reference itself too

When the caller (the parent diamond) sets the line, then the caller decides which object the call will refer to.

So lines going out of a diamond aren’t neccesarily lines set by the caller.

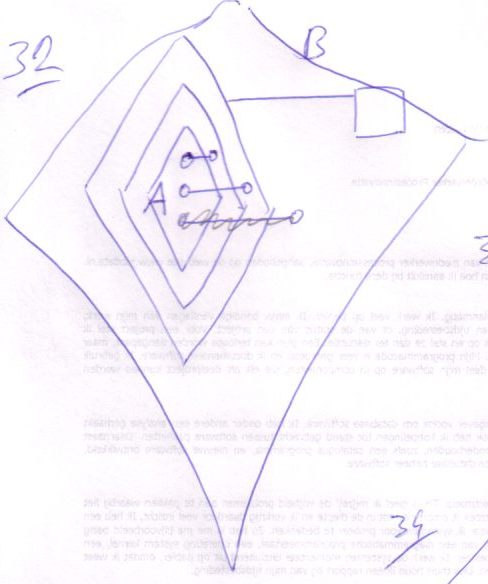
Procedures can’t set line going *into* the square themselves. Those are always set by the caller.

It’s important that the procedure itself sets lines, because the line targets of procedure members often serve as the output values of the procedure.

<I don’t know a notation to distinct sets by the caller and sets by the call. Well… in a more explicit notation you will see that the caller calls the set or the called calls the set.>

###### Clause Access

Clauses have special access privileges compared to delegated procedure symbols.



<Bad explanation following>

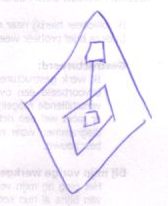
A clause can freely access anything of its parent clause and the parent clause’s parent clause, etcetera. If going up the procedure ancestry a procedure symbol is encountered with a reference line, the clause can access this ancestor freely, but not any higher in the procedure ancestry.

In the picture above, the top two lines could have been set by A itself. The first resides in A’s parent clause. The second resides in the *execution* that parents A, so still accessible to A. The last line, crossed out, can’t be set by A itself, because it resides outside A’s containing call. It can be set only by a symbol higher than A’s containing call, for instance B.

Squares formally can’t have outward lines, except for static members. There are situations in which non static members of squares are shown to have outward lines, but those are discussed later.

###### Passing a procedure reference

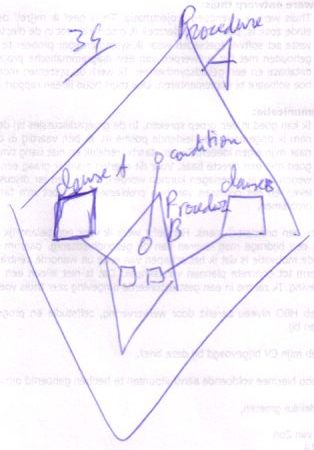
When you pass a procedure referece to a child procedure:



The child can call the referenced procedure at will.

###### The If example

A great application of that is the If procedure.



Procedure A passes Clause A and Clause B to Procedure B, along with an object that serves as the condition. Depending on the state of the condition, Clause A is executed or Clause B. So Procedure B gets to decide which clause executes.

The clauses can freely access anything of **Procedure A**, so **Procedure B** is able to call things that can freely access **Procedure A**’s stuff.

###### Brainstorm

* Simple voor de hand liggende issues regarding execution <Which precisely?>
* For instance, diamond only inside a procedure symbol
* Stuff like that. Simple issues that you can easily understand.
* Diamond pointing to another diamond
* Recursive calls... hmmm... advanced issue. Hoef je je klomp niet over te breken.

< Maar lijkt een Execution basics. Hmmm… de term Execution Basics is niet goed genoeg dan. De term basics is niet goed. Execution Facts… nah.>

* Wanneer wordt een diamond meerdere keren geexecute?

1

Diamond represents individual calls to a definition.

In a recursive situation, though: <P>, one diamond can represent multiple calls <? How to get that straight? >

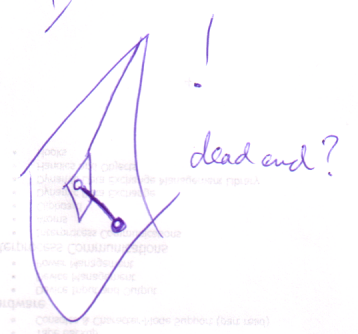
< NO IT CAN’T.>

2

A diamond pointing to another diamond: it’s a call to a call. both diamonds represent the same call. They will never execute individually.

3

If eventually the original caller is called again, with one of its calls in progress... hmm... can you call upon a call on hold, because it is waiting for one of its calls to finish? Its public objects, maybe, but you can’t execute anything in it.



4

(*Implicit* embedded procedure reference causes a separate implicit call)

5

***Procedures can only be Called from another Procedure***

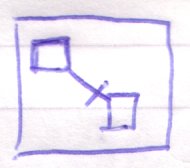
Procedures can only be called directly from another procedure.

<Because a call executes its target when its container square is executed. Actually, relate all restrictions to the fact that procedures execute.>

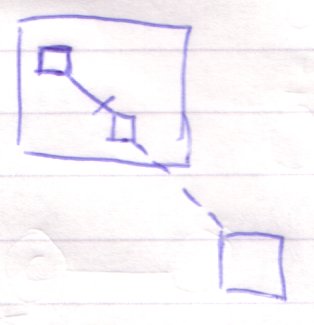
A procedure symbol can ony have a call line if it’s directly inside another procedure. Usually the call line first exits the procedure that contains the call. In other words: a call line usually directly exits a square.



Only when you call a clause or an embedded procedure reference, the line doesn’t exit the procedure first:



Call to clause



Call an internal procedure reference

Using arguments always requires assignment.

Just before execution you’ll write arguments. Just after execution, you’ll read arguments. Those are the main two occasions you’ll access arguments.

### Argument Access

##### Prolog and Epilog

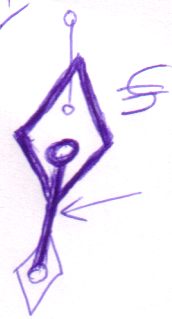
Writing and reading arguments require things to be done just before and right after the execution. Things that must be done just before execution are called the *prolog* of the execution. Things that must be done just after the execution are called the *epilog*.

Writing arguments takes place in the prolog and reading arguments takes place in the epilog.

##### Argument Assignment Requires Reading in Prolog!

Argument assignment takes place in the *prolog*. But *setting* the argument also requires *getting* the argument. However, getting the argument was said to take place in the *epilog*. For the purpose of *setting* the argument, however, *getting* happens in the *prolog*.

An assignment call:



*consults* the argument to *write*: *reads* the argument to *write*.

Assignment of arguments simply *must* take place in the prolog, even though they require reading the arguments.

Assignment procedures are fundamental procedures of the Symbol Language and behave much different from other procedures.

##### Writing Arguments

Writing an argument before execution means to set its state or line target.

An assignment as such, requires you to get a value from one place and set the argument to it. So writing an argument actually requires both a **Get** and a **Set**. The fact that it’s the argument that’s **Set** makes it writing *the argument*.

### Create Argument and Call Argument’s Members

There may be more things you want to do with an argument before passing it to a procedure. You may want to create a brand new object as an argument and you may want to call some members of the argument first. You *can’t* do these operations directly on the argument. You have to create a separate object, manipulate it and then pass it by reference

There’s a shorthand for creating a brand new object and passing it by reference. In text code this will look like this:

Procedure ( New Type )

It’s declared directly inside the procedure call. That is: unless the object is used elsewhere too. The above can also be notated as follows:

Object As Type **‘Declaration**

Object = New Type **‘Creation**

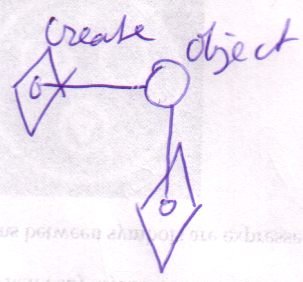
Procedure ( Object ) **‘Pass to Procedure**

But if **Object** isn’t used anywhere else, you can just type:

Procedure ( New Type )

That way, the object seems to be the argument itself, even though you’re actually passing an object by reference.

In diagram code, the declaration of the object, its creation and its being passed to the procedure, looks like this:



You can let the separate object seem to be the argument itself, by using a peel notation:



The creation will still be visible, but you’ve excluded the line, without loosing the information that it’s really an external object.

Ofcourse you don’t have to show the implicit creation.



Then you just see that the object is created, and you will be satisfied knowing only *that* it is created, not where and when. This is just about exactly the effect of:

Procedure ( New Type )

### Created Arguments

A procedure can decide if an argument is a created object or not. If it is a created object, then the object is created in the prolog, even before writing arguments. The argument object is created, then it is written, then the procedure executes.

Arguments as such are created objects directly contained by the procedure. They shouldn’t have lines.

In the epilog, first arguments are read before the created arguments are destroyed. The procedure only destroys objects it directly contains.

### Do It with a Parameter’s Members

<

*Other things done with parameters*

or keeping a reference to a created parameter or **do the same with members of parameters**… Hmmm…

Except for passing procedure reference to a procedure.

>

### Brainstorm

#### Summary of Old Subdivision

* Writing arguments before the call:
  + Assignment: lines or state
  + Creation
  + Possibly call members of an argument
* Reading parameters before the call:
  + Reading line targets (direct, final or intermediate)
  + Reading state
* Writing a parameter means setting a line or reading state from an object and writing it to a parameter
* Reading a parameter means writing it to another object
* Execution basics:
  + simple issues regarding execution
  + for instance, diamond only inside a procedure symbol
  + Stuff like that. Simple issues that you can easily understand.
* Diamond member access:
  + The caller of a diamond accesses members just before the call and just after
  + The callees of a diamond can access the caller's members if given references to them
* The terms:
  + Reference
  + call
  + definition
  + clause
  + active clause
  + inactive clause
  + sub clause / sub procedure
  + ...
* Clause access
* Differences with Com with returning object references
* Inside directly accessible, outside only 'indirectly' accessible (if line set FOR you)
* Seeming randomly referencing diamond members.
* Wanneer wordt een diamond meerdere keren geexecute?
* Diamond pointing to another diamond
* Line of connected diamonds change simultaniously

#### New Subdivision

* Parameter Access
  + Writing parameters before execution
    - Assignment: lines or state
    - Creation
    - Possibly call members of an argument
  + Reading parameters after execution
    - Reading line targets (direct, final or intermediate)
    - Reading state
  + Access during execution
    - By owner
    - By child executions (~ is this the proper place for this?)
      * The callees of a diamond can access the caller's members if given references to them
* Basic Terms
  + Reference
  + call
  + definition
  + clause
  + active clause
  + inactive clause
  + sub clause / sub procedure
  + Definitions of the terms parameter and argument
  + <Which precisely?>
  + ...
* Execution Basics
  + Simple issues regarding execution <Which precisely?>
  + For instance, diamond only inside a procedure symbol
  + Stuff like that. Simple issues that you can easily understand.
  + Diamond pointing to another diamond
  + Wanneer wordt een diamond meerdere keren geexecute?

?:

Line of connected diamonds change simultaniously

#### New Brainstorm Texts

The problems proposed in this text have to do with:

* The terms prolog and epilog, prologous and epilogous
* The term executor and execution
* Parameter assignments involve consults of parameters in prolog of procedure call, while consults are usually always in epilog.
* Diamond may only represent a single execution per definition call (not regarding unconditional jumps (? The latter is questionable.)

###### New

* Prolog and epilog
* Parameter writes in prolog
* Parameter reads (/consults) in epilog
* Assignments of parameters suggests reading parameters. However, these assignments do take place in the prolog, even though they suggest reads. Assignments are different that way.

#### Argument Access Summary

##### Argument Access

* Writing parameters before execution
  + Assignment: lines or state
  + Creation
  + Possibly call members of an argument
* Reading parameters after execution
  + Reading line targets (direct, final or intermediate)
  + Reading state
* Access during execution
  + By owner
  + By child executions (~ is this the proper place for this?)
    - The callees of a diamond can access the caller's members if given references to them

### Other

A procedure symbol can have one of four roles: - definition, - clause, - call, - reference

2004,

Generating the stack operations preceding and concluding function calls is called prolog and epilog code in C

JJ

### Procedures to Procedures, Objects to Object

< 2008-10-12 I am not sure I should impose this rule or not >

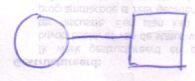
Procedures can be tied only to procedures.



Objects can be tied only to objects.

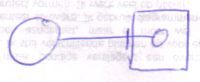


Formally, there can be no lines connecting objects and procedures:



< Ignore this talk about implicit notations. >

However, the above *is* a valid notation. It actually denotes the object being a pointer to the *return value* of the procedure. It’s an *implicit* notation (covered later), that actually standard for something else:



So then object symbols point only to object symbols and procedure symbols only to procedure symbols.

However, the picture above is actually again an implicit notation for something else, but I won’t go into detail about that yet.

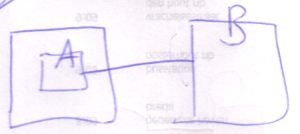
### Brainstorm

A procedure will probably not get its own symbol anyway. It would just be an object with an execution point.

A procedure symbol can define its own procedure:



But if it has a reference line it is regarded no more than a reference or call to a procedure. The target of the line is considered the procedure itself or the *definition* of the procedure.



A is the reference, B is the definition.



A is the procedure reference, B is the definition.

## More Ideas

Commands,

2009-03-13

"Do you know I don't even want those commands." ...

I made that remark when I right-clicked on an item.

The new computer language will make a ton of commands available on

many many objects... but a lot of times I don't even want those commands.

So I as the user want to be able to hide those. I do not want to see those.

JJ

##### Procedure Symbol Roles

A call line makes the square a call. A reference line makes it a reference. No line at all indicates that it’s a procedure definition or a clause. Lines can also be left out for abstraction reasons.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Definition | Clause | Call | Reference | Illegal: a square can’t be both a call and a reference |

< Not true: a call doesn’t have to directly exit a square if it calls a procedure reference embedded in the same procedure or a clause of the procedure>

Note here that the access symbol of a solid procedure reference line is also drawn solid.

##### Reference and Call Targets

Here is a list of possible reference targets and call targets:

|  |  |
| --- | --- |
| Reference to definition    Reference to clause    Reference to reference | Call to definition    Call to clause    Call to reference  (A call to a reference calls the referenced procedure) |

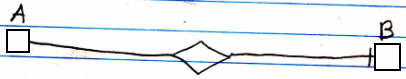
|  |  |
| --- | --- |
|  | **Reference to sibling inactive clause, in a square**  Only possible for static procedure members or exclusive establishment. |
|  | **Reference to sibling active clause, in a square**  Only possible for static procedure members or exclusive establishment. |
|  | **Call to sibling inactive clause, in a square**  This is like a jump to a clause, that would otherwise not execute.  Only possible for static procedure members or exclusive establishment. |
|  | A diamond can be executing. Only during the execution of the diamond, lines between its non static children can be real. If the diamond isn’t executing, or it’s not a diamond, but a square, lines between the non static children are suggestive. Only lines between static members in such cases, can be real. |

### From the original Assignment article group

An object assignment can also be used for *commands*. In that case it is a *command object assignment*. You can only let two command symbols refer to the same object when they are command *definitions*. You can not assign a command object to a command call. This is due to the special object creation behavior of a call. Command calls can never *redirect* their object. They are always *their own* object, and they are only created, when they are running. Command definitions, however, are permanently created objects, and a command definition symbol can redirect its object target.

To keep direct conversion between an object and a command possible without any loss of structure, it is allowed to give a call symbol and object redirection anyway. But then, the object redirection will behave as a *definition* redirection and a warning will be generated.

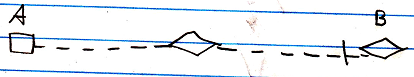
Object assignment can also be done for commands, but only has the right effect between command definitions, as they make the command definitions the same command definition object.



Class assignment is also used to assign a command definition to a call.

Command definition assignment means, that a call or command reference is appointed a new definition target. The target definition of one command reference or call is assigned as the target definition of another command reference or call. The reference to the original command definition is released. Only when the original command definition does not have any other references anymore, then the original object is deleted. But do not worry; as long as a command definition is part of a module, it means there still is a reference to it and it will not get deleted.

Command definition assignment is also a class assignment. A command definition assignment can look like this:



In the example above, symbol A is a non-executing (square) command symbol, and symbol B is a diamond, which stands for a call, but both symbol A and B cold have been either squares or diamonds.

An object pointer assignment can also be used for *commands*. In that case it is a *command reference command object assignment*. You can only let two command symbols refer to the same target object when they are command *definitions*. You can not assign a command object to a command call. This is due to the special object creation behavior of a call. Command calls can never *redirect* their object. They are always *their own* object, and they are only created, when they are running. Command definitions, however, are permanently created objects, and a command definition symbol can redirect its object target.

To keep direct conversion between an object and a command possible without any loss of structure, it is allowed to give a call symbol and object redirection anyway. But then, the object redirection will behave as a *definition* redirection and a warning will be generated.

For commands, object reference class assignment is the same as command reference definition assignment. In that case a command’s definition points to a reference to a command, instead of pointing to the command definition directly.

The notation of an assignment letting a command’s definition point to a reference to a command is the same, but then between command symbols:



In the example above, symbol A is a non-executing (square) command symbol, and symbol B is a diamond, which stands for a call, but both symbol A and B cold have been either squares or diamonds.

### Other Ideas

Commands,

2008-05-18

In the diagrams of Relations Between Commands & Objects you can see, that a call to a command can be a call upon multiple objects to execute that command. It is not yet discussed, how it is expressed that a command has access to the private contents of a class.

JJ