|  |
| --- |
| Circle Language Spec |

## Basic Diagram Elements

The diagrams might involve set of basic elements, that includes the following:

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

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

### Text

Diagrams might also contain text. Shapes might have names that could appear as text. Numbers and dates and other values might also appear in the diagram as text.

**My Object**

**My Site**

**7**

**3.141592**

**2005-08-14**

**"Hello"**

**True**

**False**

**On**

**Off**

**A = B + C**

### Circles & Triangles

Circles and triangles might represent objects, classes and interfaces. Objects may serve as prototypes or classes for other objects. That way an object might serve as the class of another object. An object might also describe the interface for another object. That may the base of the choice to have objects, classes and interfaces be represented by similar shapes.

A triangle might have a particular function related to interfaces, that may become clearer later.

### Squares & Diamonds

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

Squares and diamonds may stand for *commands*. A square may denote an *inactive* command, that might not run and could be use as a definition for other commands. A diamond might depict *execution*. A diamond might execute, while a square may not.

### Pentagons



A pentagon could represent a *module*. A module is like a little world in which objects and commands live. A module may be a site, a computer program, a library or other kind of module. A pentagon could be considered an object like circles and triangles, but a pentagon may have special purposes or behavior. The pointiness might distinguish it from other shapes.

### Nonagon



A nonagon may stand for a collection or a list of things. Objects might be placed inside the nonagon, to group them together perhaps. Maybe nonagons might not be drawn with precisely nine corners. The idea is that the pointy parts stand for multiplicity. The exaggerated pointiness might distinguish it from other shapes.

### Object Symbols

Circles, triangles, pentagons and nonagons could be called *object symbols*, since it was suggested they stand for objects.

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

### Command Symbols

Squares and diamonds could be called *command* *symbols*, since they appeared to represent commands.

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

### Lines

The idea is that, when symbols are connected with lines, those symbols have something in common.



Now follows an attempt to summarize how that might work.

### Solid Line



A solid line may point out an *object*. Symbols connected by a solid line might mean they represent the same individual object.

### Dashed Line



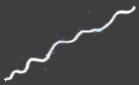
A dashed line could point at a *class* or *definition*. Symbols connected by dashed lines may be seen as having the same class.

### Dotted Line



A dotted line might point to the *interfaces*. Shapes connected by dotted lines might give them the same interface. The idea behind that is that they may look the same from the outside, but might be different on the inside.

### Wavy Line



A wavy line could represent the concept of *values*. When shapes are connected with a wavy line, it might mean, they have the same value or soon will.

### Cross

A cross might be placed inside a symbol to indicate, that an symbol is *nothing / null*: it might not refer to any object.



### Object Symbols Drawn with Different Lines

An object symbol might represent an object, a class or an interface.

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

Each object might serve as another object’s class, sort of functioning as its prototype. Any object might also provide the interface for another object, which may give another object the same exterior, while it might be different on the inside.

There is an idea for drawing the shapes with different line types, for example: dashed or dotted.

One idea is that an object only used as an object, not as a class or an interface might be drawn with a *solid* line.

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

The idea then is that if an object symbol may only used as a class, it might be drawn out with a *dashed* line.

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

An idea is also that an object symbol may only be used as an interface, then it might be traced with a *dotted* line.

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

There may be different ideas, for when it could be appropriate to draw shapes with different types of lines.

Perhaps if object symbols would be used in several ways, for instance as an object as well as other objects’ class, the symbol might get multiple borders. Here might be an object symbol that might serve as both an object and a class:



That way an object symbol could also have three borders at the same time or more.

It is not a hard rule to use different line types for symbols. But it may work intuitively to use different line types.

### Command Symbols Drawn with Different Lines

A diamond is usually an instance of a square, that represents the definition. A diamond is always seen as an instantiation, so an object. But a diamond will sometimes be used as a definition, so a class. If\* a diamond is used as a definition, it will be seen as both an object and a class, and it will get both a solid border as well as a dashed border, indicating that it is both an instance, and a definition.



When\* a diamond is not used as a definition, it only gets a solid border.



A diamond is always considered an instance, so it will always have its solid border, and is never drawn with just a dashed line:



A square is commonly used as a definition. When\* something is used as a definition, it is supposed to get a dashed line:



But\* in case of squares, this is not done, because this would clutter the diagrams with dashed squares:



Both command calls and command definitions can function as the interface for another command symbol. In that case\* the command symbol can get an extra dotted border.



A square can also function solely as an interface for a command. In that case\* it is drawn with just a dotted border:



A diamond is always considered an instance, so it is never drawn with just a dotted border:



There is an exceptional case where squares are drawn with dashed lines, but\* this is explained in a section further down.

### Connecting Object Symbols

Object symbols can be connected by different types of lines.

If\* two object symbols are connected by a solid line, it means that both symbols represent the same individual object.



Lines do have a direction, going from one symbol to the next, which will be explained later. Having a direction, an object line points out which object an object symbol will represent. The solid line is called an *object line* in this case, because it points out the object.

If\* two object symbols are connected by a dashed line, it means that both symbols have the same class, which means they have the same behavior, but\* are not the same individual object.



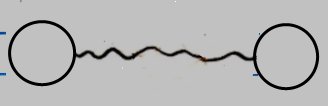
Lines do have a direction, going from one symbol to the next, which will be explained later. A class line points out, what class an object has. The dashed line is called a *class line* in this case, because\* it points out the class for the object.

If two object symbols are connected by a dotted line, it means that both symbols have the same interface, which means that they look the same from the outside, but are different on the inside.



The two symbols are separate individual objects, they also have a different class, so behave differently, but from the outside they look the same. Lines do have a direction, going from one symbol to the next, which will be explained later. An interface line points out the interface of the object.

If two object symbols are connected by a wavy line, it means the objects get the same value.



It is always about transferring a value from one object to the next. Later the expression of the direction of the line will be described.

### Connecting Command Symbols

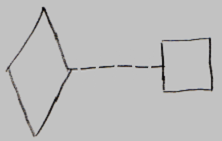
Connection between command symbols is analogous to connections between object symbols. Command symbols can be connected with different types of lines.

When two command symbols are connected by a solid line, it means, that they both represent the same command object.



Lines do have a direction, going from one symbol to the next, which will be explained later. Having a direction, the object line points out which command object the other command symbol will represent. The solid line is called an *object line* in this case, because it points out the command object. It is usually squares, that are connected by object lines, creating a command reference: a reference to a command definition. But a diamond can also get an object line, to point to another location, where the actual executable object resides.

When\* command symbols are connected by a dashed line, it means, that both commands have the same class, or definition, which means, that they have the same behavior, but are not the same individual object.



Lines do have a direction, going from one symbol to the next, which will be explained later. A definition line points out, what definition a command object has. The dashed line is called a *definition line* in this case, because it points out the definition of the command.

It is usually a diamond, an executable object, pointing out a square, the definition of the command. In that case the diamond is a replica of the definition, but it is its own individual object. Dashed lines can however be freely used to connect any two command symbols to each other, to indicate mutuality of definition.

Two commands can have the same interface, which means that one command has the same parameters as the other command. Command symbols that have the same interface are connected using dotted lines:



A dotted line is an interface line.

### Commands with Dashed Lines

In the sections above it was explained that command symbols are never drawn with dashed lines.

However, in a special case they are. When drawing out the diagram in the way explained above, the class structure of a module is drawn out with dashed lines. To also draw out the command structure with dashed lines, the *target definitions* of commands are drawn with a dashed line. What a target definition is, is explained in another article.



Also, lines that connect commands to classes are drawn with a dashed line. How commands and classes are connected is explained in another article.

It makes all structure elements and their relations drawn out with dashed lines. It is more intuitive that way: the whole class structure will be dashed, while an object structure is drawn with solid lines.

### Line Direction, Access Marks, Line Ownership

A line has direction. The direction of a line can be explicitly expressed with an access mark:

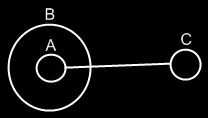


🡪

The direction goes from symbol **A** to symbol **B**. The access mark is placed right before the object, that is accessed.

But access marks are not always displayed. If you can derive the direction from a few basic rules, then an access mark is not displayed.

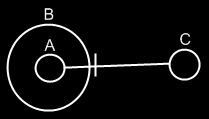
The first rule for line direction is that a line usually points outwards.



🡪

Symbol **A** has a line pointing to Symbol **C**, because lines tend to point outwards.

If the direction is not outwards, then you have to denote that with an access mark.



🡨

Symbol **C** now is` a pointer to symbol **A**.

The second rule for line direction, is that a diamond is more likely to redirect to the command definition, than it is for a command reference to redirect to a diamond.



🡪

The diamond points to the square, because it is more likely for a diamond to redirect to a square, than it is for a square to redirect to a diamond.

If the direction is the other way around direction, you will need to use an access symbol:



🡨

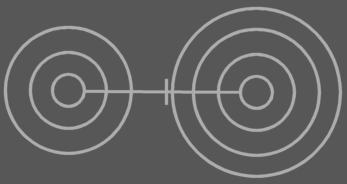
#### Access Mark Placement

When a line crosses symbols’ borders, it first exits borders and then enters borders. It is impossible to mix exits and entries.



*First come the exits a and b, and then come the entrances c, d and e.*

If an access mark is drawn to denote direction, it is by default put in the section between the exits and entrances. So not neccessarily close to the eventual symbol pointed at. In other words: the access mark is usually put in front of the border that is first entered.



If the part of the line between exits and entrances goes out of view then the access mark can be placed where it is still visible. It is placed in front of the last border in view that is entered or exited:



Do not draw it like this:



because then you are suggesting the other direction. Place it *in front* of the *last border in view* that is entered or exited.

#### Line Ownership

An object symbol can only have one object line, one class line and one interface line. The lines point *away* from the symbol. They denote which other symbol is its object, which other symbol is its class and which other symbol is its interface. Any other lines connected to an object symbol point *at* the symbol, not *away* from the symbol.



It works the same for commands. A command can have one reference line, one definition line and one interface line.

A line never gets a name. They are always called, for instance: ‘the object line of symbol B’.

### Straight Mark

A straight mark:



is used to indicate that a symbol that owns the line is **Public**, or accessible. It is also used, to indicate direction. (see above in *Line direction, access marks, line ownership*).

### Cross Mark

A cross mark:



is used to indicate, that the symbol that owns it is not accessible from the outside, or **Private**. Often a cross mark is left out altogheter, because the lack of a straight mark confirms that it is **Private**.

### Triangular Mark

A triangular mark:

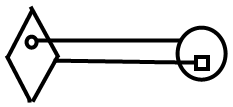


is used to indicate that a symbol that owns the line is **Protected**, meaning, that it is only accessible when the symbol that owns the line is a triangle. A triangular mark is not an arrow.

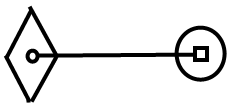
### Line Merge

Sometime two lines are so closely related, that they merge together to one line. This is called a line merge.

The lines in the following diagram are very closely related:



They merge together to one line:

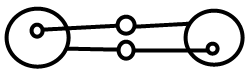


Lines merge together under different circumstances, that will come to light in later articles.

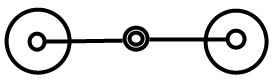
### Symbol Merge

A symbol merge is closely related to a line merge.

In the following diagram:



The lines and the symbols in circles the lines merge together:



The circles merging together is called a symbol merge. When this is applied will come to light in later articles.

## Ideas

*Below you will find loose ideas and scraps from older documentation, that are yet to be turned into good documentation.*

Basic Diagram Elements,

2009-08-18

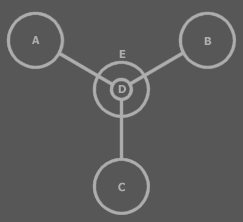
The first diagram is ineffective. ‘It could be an elephant.’ Find a more concrete example. Something that is something for real.

JJ

# From the original Symbol documentation

#### Direction

Lines tie symbols together, saying that they share a certain aspect.



A, B, C and D share an aspect: they represent the same object. E represents another object.

In that sense the lines have no particular direction. However, the direction of lines does matter as will become apparent in later subjects. It is of the essence to see that one symbol is pointing to the other. There are rules that determine the direction of lines.

##### Access Symbols

A *line dissector*, also called an *access symbol*, can determine the direction of a line. It sort of denotes which symbol is pointed at:

|  |  |
| --- | --- |
| B is pointed to.  The direction goes to the right 🡪.  In this case A points to B. | A is pointed to.  The direction goes to the left 🡨.  In this case B points to A. |

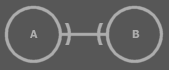
You can see the line dissector as the door that lets you access a symbol. Therefore in the first example, **B** is accessed: **B** is pointed to by **A**, and in the second example **A** is accessed: **A** is pointed to by **B**.

In another situation the access symbol is placed as follows:

|  |  |
| --- | --- |
| The direction goes to the right 🡪.  A points to B. | The direction goes to the left 🡨.  B points to A. |

In the last picture, the access symbol isn’t placed near **A**, but closer to the border of **C**. Why this is so is explained later. For now you can see the line dissector as the *door* to **C** that lets you access **A**.

The access symbol can also be put at both ends:



In that case the direction goes both ways. Actually, there’s two lines: one going one way and the other going the other way.

When an access symbol is left out:



Three things can be the matter:

* The direction goes *both ways*
* The direction doesn’t matter in this view
* Other rules determine the direction

Those other rules will be covered next.

In the example above, though, no other rules determine the direction.

##### Precedence of Direction Rules

There is a precedence of rules that determines the direction of a line.

###### Access Symbols

The first rule implies that an access symbol determines the direction of a line. If other rules can determine the direction, the access symbol is usually left out. This means that if other rules *can’t* determine the direction, an access symbol is used.

The other rules determine the direction without the use of an access symbol. These rules are based on which direction is most common. The more common direction doesn’t require an access symbol. The less common direction requires an access symbol. These rules are invented to as little as possible disturb the diagram with access symbols.

Because an access symbol is decisive for the direction, the access symbol rule can be regarded the *first* rule. However, only if the other rules can’t determine the direction, an access symbol is used. In that sense it is the *last* rule: the last means for denoting direction.

Don’t go numb on the rules that follow. They only serve the following purpose: common situations don’t require an access symbol to determine the direction.

###### Outward

This rule applies when the access symbol rule doesn’t override it. If the access symbol is left out then the direction is outwards:

Direction is Usually Outwards



A points to B, because the direction is outwards.

You can see **A** as being **C**’s *eye* to **B**, so it is logically directed outwards.

###### You Sooner Exit a Procedure than an Object

The rule that follows only applies if the direction isn’t determined by the two rules above: the access symbol rule or the outward rule.

You Sooner Exit a Procedure than an Object

It happens more often that a procedure has a pointer to an object:



🡪

than for an object to directly reference a procedure parameter:



🡨

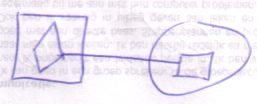
That is actually highly uncommon. It’s *not* uncommon to reference a procedure’s object, but it *is* uncommon to reference it directly from an object.

It is very *common* for a procedure to have a reference another object’s procedure:



🡪

Especially when it’s a diamond:



And it is very *uncommon* to reference a procedure clause from an object:



🡨

Altogether it is more common for a procedure to point to something in an object, than for an object to directly reference something in a procedure: you sooner exit a procedure than an object.

###### You Sooner Reference an Interface than Redirect It

Or: you sooner point *to* a triangle than *from* a triangle. This rule only applies if the rules above don’t determine the direction: access symbol, outward, sooner exit a procedure than an object.



🡪

The direction goes to the right 🡪. It’s more common for a circle to redirect to an interface implementation in another circle. The other direction is less common: it’s less common to redirect an interface implementation to some far away object.



🡨

In this example the direction *is* to the left 🡨. You *do* sooner reference an interface than redirect an interface, but the outwards rule is still dominant. In the first diagram the outwards rule didn’t apply; here it does.

###### You Sooner Redirect a Diamond than Reference It

A diamond is usually a call, so it’s most common that the diamond points at something:



🡪

You can point *to* a diamond:



🡨

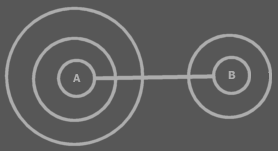
But it happens to be so that a diamond usually points *at* something. The picture above shows a reference to a diamond symbol.

###### Exit the Most Borders

This rule is derived from the outwards rule. What they have in common is:

You sooner exit than enter.

The outwards rule says that you exit a border rather than enter it. The exit the most borders rule says you exit more borders than you enter.



A points to B becase because you exit more than you enter.

This rule only applies if no other rule has already determined the direction. In many cases the rule is ignored, and an access symbol is used, but there *are* situations in which it’s common to determine the direction from this rule, and there an access symbol is often left out.

###### In Short

So not looking at the precedence of rules, the rules can be viewed as follows:

- Access symbol is decisive

- You sooner exit than enter

- You sooner exit a procedure than an object

- You sooner reference an interface than redirect it

- You sooner redirect a diamond than reference it

##### Bidirection

If *no rule* determines direction then the direction is either not expressed in the diagram or the direction goes both ways.

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

The direction *certainly* goes both ways if both ends have an access symbol:

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

A line going both ways is called a *bidirection*. Formally there are actually two lines: one for each direction, but only one line is shown.

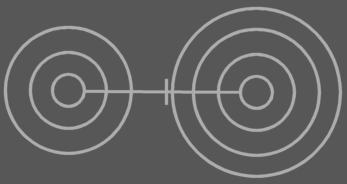
##### Access Symbol Placement

When a line crosses symbol borders, it first exits borders and then enters borders. It’s impossible to mix exits and entrances.



First come the exits a and b, and then come the entrances c, d and e.

If an access symbol is drawn to denote direction, it is by default put in the section between the exits and entrances. So not neccessarily close to the eventual symbol pointed at. In other words: the access symbol is usually put in front of the border that is first entered.



If the part of the line between exits and entrances goes out of view then the access symbol can be placed where it’s still visible. It is placed in front of the last border in view that is entered or exited:



Don’t draw it like this:



because then you’re suggesting the other direction. Place it *in front* of the *last border in view* that is entered or exited.

###### Overview of Access Symbol Placement

* If a line crosses borders it first exits borders and then enters borders. You can't mix exits and entrances.
* If an access symbol is drawn to denote direction it is by default put in the section between the exits and entrances, so not neccessarily with the eventually entered symbol.
* In other words: the access symbol is usually put in front of the border that is first entered.
* If this access symbol will go out of view then the access symbol can be placed where it’s visible.
* It is placed in front of a border that is gone entered or exited.

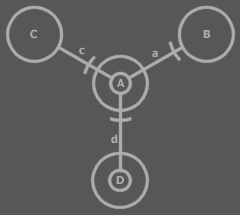
#### Line Ownership

##### Object Symbol Lines

Direction is important in understanding that one object symbol can have only one line of each type.



An object symbol can only have one object line, one type line and one interface line. The lines point *away* from the symbol. They denote which other symbol is their object, which other symbol is their type and which other symbol is their interface. Any other lines connected to an object symbol point *at* the symbol, not *away* from the symbol.



**a** points *away* from **A**. This is **A**’s line. The other lines: **c** and **d**, point *to* **A**. In other words: **C** and **D** are pointers to **A**. **A** is a pointer to **B**.

##### Procedure Lines

It works the same for procedures, except that procedures have reference and interface lines.

#### Object Basics

##### Notation Elements



Circles and triangles represent objects. For that they are called *object symbols*.

A circle is the usual symbol for an object. A triangle is a special symbol, as I explained earlier and will elaborate on later.

When I say *symbol* in this section, I’m often talking about object symbols.

Relations between symbols are expressed by *containment*:



and by connecting them with *lines:*





Dotted lines, dashed lines and solid lines.

##### Objects

Each symbol represents an object:



You can make two symbols represent the same object by connecting them with a line:



A solid line in this case is also called an *object line*, because it denotes which symbols are the same object.

##### Types

Each symbol is a type:



You can make two symbols be of the same type by connecting them with a dashed line:



Both symbols are of the same type, but they are separate objects.

Every symbol of the same type has the same contents. The type can be edited by editing either symbol and their contents will change simultaniously.

A dashed line in this case is also called a *type line*, because it denotes which symbols are the same type.

## Executions & Parameters

### Brainstorm

#### Other: Might contain a good text for justification of a diamond symbol rather than a call line

**This IS the text that lead me to wanting a diamond symbol instead of a call line.**

<Why the hell not, actually? Ok, you can’t do it in other languages, but why the hell can’t you do it here. Oh. When you reference a call, it makes the call line function as a reference line rather than a call line. If you want this to be different, a reference to call would become a call, which is not something you want to happen in your system. The reference target would get control over if the source will be a reference or if the source will execute. The source would have no say in that. Unacceptable. So a candidate for an alternative rule for A Call Can’t be Called or Referenced is: if you reference a call, its call line is treated as a reference line.

If you call the reference to a call…

Een call line is eigenlijk een reference line, maar als de parent square execute, dan execute ook de call. Hmmm… het is bijna of het allemaal reference lines zijn en sommige squares executen nou eenmaal. Shit! Dat is ook zo! Clauses moeten ook kunnen executen en ik zeg nu dat iets alleen execute als het een call line heeft!!! Hmmm… shit, een call is een tag, niet een line!!!!!!!!!!!!

Een call een aparte line maken is net zo iets als een triangle een aparte line maken. Hetzelfde mankement. Dan lijkt het logisch, dat een call een apart symbool krijgt, niet een aparte line. Shit. Welk symbool.

## When Shape Types, When Line Types

<So there’s a call trace and an definition trace.>

<Target call, target definition>

<Misschien mag ik dan al wel verklappen dat de call zo’n beetje het object is en de definition de klasse.

En waarom het dan aparte symbolen zijn en niet verschillende line types. Eigenlijk moet ik dan de keuze voor alle line types en symbolen aangeven.

Je kunt maar 1 reference line hebben. Dat is zo’n beetje de reden. Hè, ik moet het inderdaad goed opschrijven

Call is een hoedanigheid van het symbool, dat geen line behoeft.

Type is niet een hoedanigheid van een symbool, het is waar een symbool voor *kan* dienen.

Een triangle is gekozen voor interface implementation, omdat een interface implementation ook een hoedanigheid is van het symbool, dat geen line behoeft.

Eigenlijk is het wel een beetje zo dat: alles wat ik aan mezelf heb moeten uitleggen, moet ik aan de lezer uitleggen.

>

A definition is a lot like a type of procedure, while an execution is an instance of the procedure. As I state this relation, it may seem strange to you that I picked *separate symbols* to denote a procedure symbol’s execution and definition, while for objects I use different *line types* to separate objects from types. Diamond is actually the extra symbol picked to represent an execution. An execution of a definition is like an object of a type. However, an execution has another special characteristic: it executes.

When an object symbol has a type line it’s behavior in the container isn’t as much different as

However, object symbols aren’t different to their container if they

To find the definition you follow the reference line between

Just consider: if a diamond doesn’t have a line it is an executing clause, when a square doesn’t have a line, it’s a non executing clause. In both cases it’s a definition. But the two case differ in that in one case it executes and in the other it doesn’t. If an object symbol has no line, it’s an object. Simply stated, it needs an object line for it not to be a type. A procedure symbol shouldn’t need a line to be an execution.

When a characteristic’s presence shouldn’t be dependent of the presence of a line, it needs to be drawn out with a shape type. If a characteristic is dependent of the presence of a line, it’s the line presence that gives it the characteristic. If I’d want object symbols to serve only as a type and not as an object, then I’d need to reserve a special shape to separate types from objects. Now, to make an objet symbol serve as a type only and not as a type, I make the Object Get Inaccessible? NO. That’s not true. Actually I’d have to not be able to Symbol Get if it’s for the purpose of assigning an object line.

A square is never an object.

For a procedure symbol to function as a reference and not as an execution is not up to the possibility to have a symbol as an execution target.

Ok, if I wanted it so that an object symbol could only function as a type, but not as an object, I should reserve a separate shape for it. But that doesn’t mean that an object shape can’t serve as a type. An execution shape can serve as the definition too.

It’s important for some procedure symbols not to function as an execution. Otherwize the system would behave complete different. It’s not as important for an object symbol not to function as an object, only as a type. The system isn’t really harmed as severly by that. The same goes for triangles: if a triangle is suddenly a circle, the system behaves completely differently.

I’m still in doubt. I think it’s good that there’s a diamond symbol and that there aren’t separate object, type and interface shapes. I just can’t define *why* yet.

It totally makes sense to use separate type, interface and object shapes, but … it’s just not that important. Not as important as the function of diamonds, triangles and pentagons.

Sure it is nice to see in a system that one set of object symbols serve as the types… So it might be an idea to make it possible to give them a different shape type… when you can’t use the symbol as an object target.

The total reason of the diamond and triangle symbols is that the effect of it has greater consequences for the behavior of the system. A diamond symbol makes rules easier to understand: it’s easier to get: “you can’t place a diamond in an object symbol”, than it is to remember “a procedure symbol in an object symbol can’t have a call line”

*“The reason why both call and reference lines need to be followed is because call and reference lines are actually both kind of like reference lines. However, a call line has the side effect that its square will execute if its parent square executes.”*

Dat is zo’n beetje waar het kwartje begon te vallen dat het een shape moest zijn en geen line type.

Een andere shape wordt in basis Symbol alleen gebruikt als het echt nodig is. Als het niet echt nodig was om andere shapes te gebruiken, dan maakte ik het *allemaal* cirkels.

## Brainstorm

* Element combining
  + Containment
  + Linkage
  + Positioning
* A symbol’s line points *away* from the symbol.
* The other connected lines point *to* the symbol.

Notation Methods versus System Rules

Implicit calls are but notation methods, that don’t affect the behavior of the system. Type genericity, interface genericity and type interface genericity are system rules. They affect the behavior of the system.

2004,

Every circle or triangle represents an object.

JJ