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

## Basic Diagram Elements

This article attempts to give a more or less formal overview of elements that might appear in the diagrams. For some reason the choices for the different shapes is sometimes commented on too.

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

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

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

### Text

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

**MyObject**

**My Site**

**7**

**3.141593**

**2005-08-14**

**"Hello"**

**True**

**False**

**On**

**Off**

Perhaps some of the code might not be visualized graphically, but just displayed as text expressions:

**A = B + C**

### Circles & Triangles

Circles and triangles may represent objects, classes and interfaces. An object might serve as the class or prototype of another object. An object could perhaps also describe the interface for another object. A choice was made to have objects, classes and interfaces be depicted by similar shapes.

A triangle might have a particular function related to interfaces, which may be clearified later. Circles may be more common than triangles.

The choice for having a circle depict an object, might relate to the view that objects could be considered the foundation for object oriented programming, and circles may arguably the most basic shape there is. So the most basic shape would then be used for the most basic concept.

### Squares & Diamonds

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

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

The choice for a square may be while finding the second most basic shape apart from the circle, trying to represent possibly the second-most basic concept of object oriented programming: procedures. The diamond came out of the desire for a shape related to squares. Somehow there is the idea, that it would look like an 'activated' square in a way.

### 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 just like circles and triangles, but a pentagon may have special purposes and behavior. The pointiness might be to distinguish it from other shapes, which may help visually perhaps. It sort of looks like a house maybe.

### 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 be to distinguish it from other shapes. It also might sort of make it look like the inverse of a circle: a circle might stand for *one*, while an inverted circle may depict the idea of *many*.

Thoughts while looking for shapes to symbolize things might have been: As shapes become bigger, it may not be easy to distinguish hexagons, septagons, octagons and nonagons. Less than nine corners, may lead to shapes that intuitively may symbolize something else: Hexagons: Do bees have anything to do with it? Septagons: Maybe confused with pentagons? Octagons: A stop sign? should I stop?

### 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 would be connected with lines, those symbols would 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 symbolize the same object.

### Dashed Line



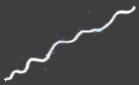
A dashed line could point out a *class* or *definition*. Symbols connected by dashed lines could be thought of as having the same class.

### Dotted Line



A dotted line might point out an *interface*. 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 behave differently 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 perhaps soon will get the same value.

As we might be running out of different ways to draw lines, this wiggling of the line may symbolize heat or energy. Values might regularly flow from one place to the next. So a symbolization of something active like waviness seemed appropriate.

### Cross

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



### Object Symbols Drawn with Different Lines

An object symbol might symbolize 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.

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

If an object symbol might only used as a class, it might be drawn out with a *dashed* line.

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

An idea is also that if an object symbol may only be used as an interface, 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 another object’ class, the symbol might be given multiple borders. Here might be an object symbol that could serve as both an object and a class:



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

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

### Command Symbols Drawn with Different Lines

The idea for symbols drawn with different line types might also be applied to *command* symbols.

A diamond, which might represent a command call, may regularly be an instance of a square, that could represent a command definition. When the diamond is an instantiation, it might be seen as somewhat of an *object*. But a diamond might sometimes be used as a definition, more like a *class*. When a diamond is used as a definition, it might be seen as both an object and a class, so it might be given both a solid border as well as a dashed border, indicating that it might be both an instance, and a definition.



If a diamond would not used as a definition, it might only get a solid border.



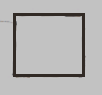
It might be that a diamond would regularly be considered an instance, so it might often have just a solid border. It may become uncommon for it to be drawn with just a dashed line:



A square may be used often as a definition. The idea was proposed that if something may be used as a definition, it might be drawn with a dashed line:



A choice could be to not do this, because otherwise you might mostly see dashed squares,



It depends the choices of how to use this notation of borders with different line styles.

There may be a particular case where you might draw a square with a dashed line, but the suggestion for this may have to wait.

Both command calls and command definitions may function as the interface for another command symbol. In that case the idea is that the command symbol may get an extra dotted border.



It could be an idea that if a square functions solely as an interface for a command, the square might be drawn with just a dotted border:



It is suggested here, that a diamond might so often be thought of as an instance, that you might choose to always show the solid line and not draw it with just a dotted border:



### Connecting Object Symbols

Object symbols could be connected by different types of lines.

When two object symbols would be connected by a solid line, it might mean that both symbols represent the same object.



One symbol might be considered to represent the actual object while the other symbol might represent an *object reference*. An object line may point out which object an object symbol would point to. The solid line might be called an *object line* in this case, because it may point out the object.

When two object symbols would be connected by a dashed line, it might mean that both symbols may share the same class, which could mean they may have the same behavior, while they might not represent the same individual object.



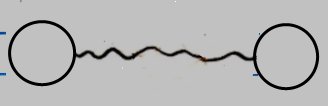
A class line may points out, what class an object might have. The dashed line might be called a *class line* in this case, since it would point out a class for the object.

If two object symbols would be connected by a dotted line, this might depict that the symbols would have the same interface, which may mean they could look the same from the outside, but might be different on the inside.



Then the two symbols might be two separate individual objects, they may also have a different class, so could behave differently, but since they seem to have the same interface, this might mean that from the outside they might look the same. An interface line might point out the interface of the object.

When two object symbols would be connected with a wavy line, it might mean the objects would get the same value.



This might express that the values of the two objects are the same. It could also mean transferring a value from one object to the next.

### Connecting Command Symbols

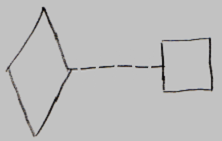
Connection between command symbols could be parallel to connections between object symbols. Command symbols might be connected with different types of lines.

When two command symbols would be connected by a solid line, it might mean that they would represent the same command object.



The object line might point out which command object the other command symbol would represent. The solid line might be called an *object line* in this case, because it might be used to point out the command object. It might be more common for squares to be connected by object lines than diamonds. A square connected with an object line might depict a command reference: a reference to a command definition. A diamond migh also get an object line, that may point to another location, where the actual executable object may reside.

Command symbols might be connected by a dashed line. This may be used as indication that both commands would have the same class, or definition, which may mean, that they would get the same behavior, but might not be the same individual object.



This definition line might point out, what definition a command object would have. The dashed line could be called a *definition line* in this case, because it seems to points out the definition of the command.

It may be more common for a diamond, an executable object, to point out a square, the definition of the command. In that case the diamond might be a replica of the definition, but it could that it would be its own individual object. Dashed lines might however be freely used to connect any two command symbols to each other, to indicate mutuality of definition.

Two commands could have the same interface, which might mean that a command has parameters similar to the other command. Command symbols that would have the same interface might be connected using dotted lines:



A dotted line could be an interface line.

### More Suggestions for Dashed Lines

#### Commands with Dashed Lines

Previous text tried to show that command symbols might scarcely be drawn with dashed lines.

However, in a certain case it might be drawn with a dashed line. When drawing out the diagram in a way demonstrated above, the class structure inside a module could be drawn out with dashed lines. To also draw out the command structure with dashed lines, the *target definitions* of commands might be drawn out with a dashed line. Later text may show what a target definition might be.



Lines that connect commands to classes might be drawn with a dashed line.

#### Object Structure Solid, Class Structure Dashed

Drawing dashed lines this way might make the structure elements and their relations be drawn out with dashed lines. It might be more intuitive that way: the class structure would be dashed, while an object structure would be drawn with solid lines.

#### Other Ideas

However, there might be different ideas of how to apply dashed lines, for instance one where dashed lines are reserved for the expression of the concept of *static*.

### Line Direction, Access Marks, Line Ownership

A line might have direction. The direction of a line could be expressed with an access mark:



🡪

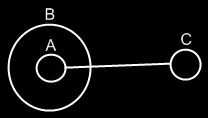
The direction would go from symbol **A** to symbol **B**. The access mark might be placed before the object, that would be accessed.

Admittedly, there is the idea to just draw it as an arrow shape, but currently access marks like this are suggested.

But access marks might not always be displayed. Some suggestions are made here for when direction might be implied without the use of an access mark

#### Lines Pointing Outwards

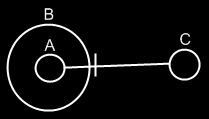
The first suggestion for line direction is that a line might usually point outwards.



🡪

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

If the direction would not be outwards, it might be denoted with an access mark.



🡨

Symbol **C** would then be like a pointer to symbol **A**.

#### Diamonds More Likely to Redirect

The second suggestion for line direction might be that a diamond might be considered more likely to redirect to the command definition, than it would be for a command reference to redirect to a diamond.



🡪

The diamond would point to the square, when it would be considered more likely for a diamond to redirect to a square, than for a square to redirect to a diamond.

If the direction would be the other way around direction, it might be denoted with the use of an access symbol:



🡨

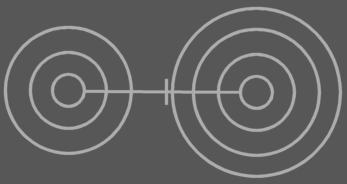
#### Access Mark Placement

When a line crosses symbols’ borders, it might first exit borders and then enter borders.



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

An idea is to put the access mark in the section between the exits and entrances, not neccessarily close to the eventual symbol pointed out. In other words: the access mark would then be placed in front of the border that may first be entered.



Perhaps if the part of the line between exits and entrances would go out of view then the access mark might be placed where still visible. It could be placed in front of the last border in view that might be entered or exited:



It may be a problem to draw it like this:



This might suggest that the rightmost border would be entered, instead of exited.

#### Line Ownership

Perhaps depending on how these diagrams might be applied, an object symbol could only have one object line, one class line and one interface line. The lines may be considered to point *away* from the symbol. They could denote which other symbol may be its object, which other symbol may be its class and which other symbol may be its interface. Other lines connected to an object symbol could be considered to point *at* the symbol, instead of *away* from the symbol.



It might work similarly for commands. A command might have just one reference line, one definition line and one interface line.

### Straight Mark

A straight mark:



might be used to indicate that a symbol that might own the line would be **Public**, or accessible. (It is supposed to be about the smaller line, that appears to dissect the longer line.)

The straight access mark might be connected to a symbol as follows:



It might also be used, to indicate direction. (See above in *Line Direction, Access Marks, Line Ownership.*)

### Cross Mark

A cross mark:



might be used to indicate, that the symbol that might own the line would be not accessible from the outside, or **Private**.

Here is how it might look when connected to a symbol:



A cross mark might be left out, if the choice is made that the absence of a straight mark might say it would be **Private** already, but there could be different ideas about that.

### Triangular Mark

A triangular mark:



may be used to indicate that a symbol that might own the line would be **Protected**. Protected is something that can play a role in the concept of *inheritance*. In the Circle language it might mean that a symbol might only be accessible if it was placed inside a triangle.

Here is how that might look if it is connected to a symbol:

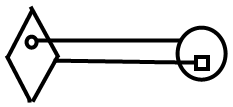


A triangular mark would not represent an arrow. It might be confusing that a triangular mark looks like an arrow, but isn't, and a different notation for protected might be appropriate and introduced some time.

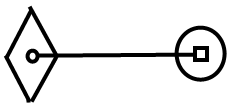
### Line Merge

A suggestion is made here to sometimes merge two lines together to form one. The idea is that the two lines would be so closely related, that they might merge together to form one. This might be called a line merge.

The lines in the following diagram might be very closely related:



They may merge together to one line:



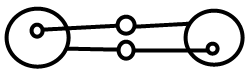
Lines might merge together under different circumstances, that may come to light in other articles.

There may be some reservations about using this notation, because it might introduce some ambiguity or the suggestion of ambiguity, but the idea may still appear in some of the text.

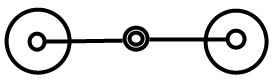
### Symbol Merge

A symbol merge would be closely related to a line merge.

In the following diagram:



Some lines and circles might merge together:



The idea of circles merging together might be called a symbol merge. How this might be applied may come to light in other articles. It could be applied to the concept of *automatically synchronized bidirectional relationships*, if you happen to have an idea of what those might be.

## Ideas

*Below you may find loose ideas and parts from older documentation, that are intended to be turned into more polished documentation.*

Basic Diagram Elements,

2009-08-18

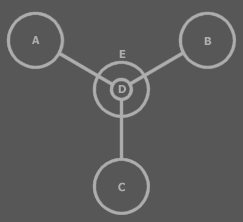
The first diagram is ineffective. ‘It could be an elephant.’ Find a more concrete example. Something that is something for real.

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. |

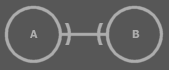
You can see the line dissector as the door that lets you access a symbol. Therefore, in the first example, **B** is accessed: **B** is pointed to by **A**, and in the second example **A** is accessed: **A** is pointed to by **B**.

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:



🡨

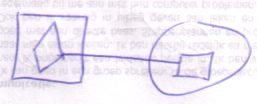
That is actually highly uncommon. It’s *not* uncommon to reference a procedure’s object, but it *is* uncommon to reference it directly from an object.

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:



🡨

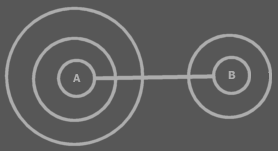
But it happens to be so that a diamond usually points *at* something. The picture above shows a reference to a diamond symbol.

###### 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 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.

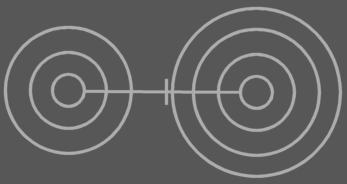
##### Access Symbol Placement

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 necessarily 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 necessarily 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.

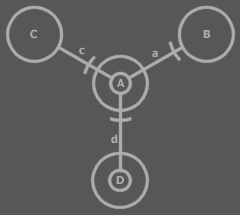
#### Line Ownership

##### 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 simultaneously.

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 zoiets 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 object 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