|  |
| --- |
| Circle Language | Construct Drafts |

## Basic Diagram Elements Construct Drafts

### Loose Ideas

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

* 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.

#### Brainstorms Shape vs Line Type

This might be (possibly complicated) text for justification of a diamond symbol rather than a call line. It also seems sort of a mystery when something is symbolized with a shape, and when by a line drawn with a certain style.

If there is nothing to point to, while there is an aspect to express visually, that may exclude a line from being used and may 'require' a shape, but then why one time the same shape with a different line type versus a different shape? The choice 'different shape' vs. 'shape drawn with different line style' seems not easy to rationally pinpoint. Maybe it could be used as food for thought when 'abusing diagram expression' might be worked out some day perhaps. Then the idea was a systematic overview all methods of symbolizations used in Circle vs possibility to express aspects of systematics using that symbolization.

##### Executions & Parameters Brainstorm

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

<Why not, actually? Ok, you can’t do it in other languages, but why 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. O! Dat is ook zo! Clauses moeten ook kunnen executen en ik zeg nu dat iets alleen execute als het een call line heeft!!! Hmmm… o, 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. O. Welk symbool?

##### When Shape Types, When Line Types

<So, there’s a call trace and a 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.