|  |
| --- |
| Circle Language Spec: System Objects |

## System Objects Ideas

*The texts below are material, yet to be turned into good documentation. They are categorized into subjects and each subject may become a separate article in the future. They are ordered roughly by importance.*

### More Commands & Aspects

Final targets,

2004

Another group of aspects is the *target aspects*. The target aspects of an *object symbol* are:

* + Target Object
  + Target Type
  + Target Interface

And those of a *procedure symbol* are:

* + Target Definition
  + Target Execution
  + Target Interface

These are derived from the line aspects. By doing a trace, the target aspects are retrieved.

They are commands, that may belong to another aspect.

> 2009-08-31 Do not forget the difference between target definition and target execution.

JJ

-----

> You should complete the list of aspects with other aspects, that control the system objects, such as the Command aspect, saying whether it is a command, but that could / should be done later.

-----

Computer Language,

System Objects

Je moet ook system commands hebben, die een ->1 related item toevoegen, en dergelijke. Je moet eigenlijk alles van het diagram kunnen genereren. Ik ben geen voorstander van code generators,

maar het moet wel mogelijk zijn.

JJ

-----

- Komt later: onderscheid Get For Copy en Get For Access

> Accessing an object requires an Object . Get

> Object assignment requires an Object . Get and an Object . Set.

-----

What happened to commands like Target Object Get?

-----

- Get Purposes

- Get

- Copy

- Access

- Redirection

- Use as Object

- Use as Type

- Use as Interface

- Use as Procedure

- Use as Procedure Interface

-----

<How about system aspects that return how many exits a line makes and which symbols it enters?>

> 2009-09-06: I do have to do something with qualified references, though.

-----

Symbol

Get : Copy

Object

Get : Copy or Redirection

Set

Type

Get : Copy or Redirection

Set

Interface

Get : Copy or Redirection

Set

State

Get : Copy

Set

Target Object

Get : Copy or Access

Target Type

Get : Copy or Access

Target Interface

Get : Copy

-----

When executing a Target Object Get or Target Type Get, automatically are performed Object Redirection Gets and Type Redirection Gets. <other place>

-----

So Gets can have three purposes: *Copy*, *Access* or *Redirection*. *Copy Get* happens when a programmer stores the gotten value somewhere else using a Set. *Access Get* happens when the programmer accesses a member of an object. This only happens for Target Objects, and sometimes Target Types too for static elements. *Redirection Get* happens when tracing the target object, target type and target interface. Each redirection step in this process requires getting the symbol to which there’s redirected. Redirections are never directly initiated by the programmer, but are done automatically.

Every system aspect supports a different collection of get purposes. For instance you can’t get the Object for *access*. You can only get the *Target* *Object* for access. The *Object* you can only get for copy and redirection.

When the main purpose of a Get is mentioned, the word ‘Get’ is often left out. You can simply speak of for instance Object Copy or Type Redirection.

The reason that this separation between get purposes is made is not that there are three different get procedures. The get procedure of all three purposes is the same procedure so that they return consistently the same value for all three purposes. What you can control separately for the three purpose is their access controller (not covered yet). Redirection is always public: the Symbol system can always redirect whenever it feels like it, because Symbol knows perfectly when to redirect and when not to. You *can* put restrictions on Access and Copy. If you make Access private then you can’t access the symbol (through its parent). If you make Access public then you *can* access the symbol. If you make Access public, but Copy private it means that you *can* access the symbol, but you *can’t* copy the id of the symbol. To be able to do that, Copy needs to made public. Copy is always equally or less public than Access. One of the main purposes to different access controllers for different Get purposes is to be able to access a symbol, but at the same time not be able to copy the reference to the symbol. The other important purpose is that the Get always has to be public for redirection, even when it’s private for other purposes. You can alter the access controllers of almost every system procedure and purpose, except for Redirection. That one’s always public. Public for the Symbol system that is, because programmers can never call this form of get themselves.

-----

Just now, I’ve only discussed the system procedures of *object symbols*. Procedure symbols have system procedures too:

Symbol

Get : Copy

Call

Get : Copy

Set

Reference

Get : Copy or Redirect

Set

Target Procedure:

Get : Copy or Access

*Symbol Get* obtains the id of the symbol itself. *Call Get* obtains the call line target symbol. You can also *Set* the call target. *Reference Get* returns the symbol id of the reference line target. You can also *Set* the reference line target. The *Target Procedure* of a symbol returns the procedure at the end of the redirection: the procedure definition. *Target Procedure Copy* copies the id of this procedure. *Target Procedure Access* is the Get that occurs when you call the target procedure. Access controllers can be applied to all elements, except Symbol Get. Call and Reference Get and Set can be redefined. Mind that the access controller for Target Procedure Access determines *the* access controller for calling the procedure symbol.

-----

System Objects,

2009-08-31

You might create something to only retrieve related commands or only related (non-command) objects, but that is a detail.

JJ

-----

System Objects,

2009-08-31

In 2004 there used to be the Shape aspect, but now I would make them a set of booleans:

* Is Command
* Is Executable
* Is Interface
* Is Module

JJ

-----

System Objects,

2009-08-31

What represents the state of an object, are all the values stored by the object, including the ID’s of related objects, etcetera. It might also be callled an aspect, but is not to be confused with the Value aspect.

JJ

-----

System Objects,

2004

The State symbol aspect can be obtained and assigned too.

- State Get

- State Set

The state is obtained to be immediately assigned to another object. That way the values and line targets of one object are copied to another object, giving both objects the same state.

> 2009-08-31: so also, the targets of the sub-objects would be copied… or the targets of a related item.

JJ

-----

System Objects,

2009-08-31

Can the Related Items and Related Lists be called a separate aspect, such as the Ancestry aspect or something similar?

JJ

-----

System Objects,

2009-08-31

In 2004 I had the Symbol aspect, that returns the symbol itself. That is now the Reference aspect. I had already thought of that in 2004.

JJ

-----

System Objects,

2009-08-31

Should I mention the inseparability of Get and Set?

A Get is always accompanied by a Set?

Get For Access is different then, though…

JJ

-----

System Objects,

2009-08-31

Get for Access:

Get it for a special purpose, called Access, described below.

JJ

-----

System Objects,

2009-08-31

‘Get Purposes’

Perhaps these are supposed to be called ‘Get Purposes’:

- Use Reference As Object

- Use Reference As Class

- Use Reference As Interface

- Use As Class

- Use As Interface

- Get For Access

- Get For Copy

> They are all Object Get or Reference Get.

JJ

-----

System Objects,

2009-08-31

I forgot about annulling the Class. This removes the class list completely. How do I express that?

JJ

-----

System Objects,

2009-09-03

Can you display an Access call explicitly?

JJ

-----

I forgot that a Related List also has the Name aspect.

### Extension of System Commands

The Get usually returns system information. The Set usually changes a line target normally. However, as a programmer you can also decide for yourself what’s the effect of a Get or a Set. You can totally redefine the Get and Set procedures of a symbol. With that you can also add parameters to the Get and Set procedures. So you’re able to decide yourself which symbol to return on a Get and which symbol to assign on a Set. \* You can also for instance see to it that on a Set call, the change is applied to all sorts of things, by starting procedures that apply the new value. When setting an object’s state, for instance, you can automatically apply its new state to other objects, for instance when you need to draw a text on screen every time a value changes. You can’t redefine *all* system procedures. For instance the effect of Target Object Get must stay consistent and can not be customized. And also Symbol Get always needs to return the symbol itself. You can only redefine the Object, Type, Interface and State Gets and Sets.

-----

System Objects,

2004

- Setting lines

- default implementation

> 2009-08-31: Daarmee bedoel ik: wat opgeslagen is bepaalt de line target

- What get returns determines line.

> 2009-08-31: How does that look?

> 2009-08-31: Daarmee bedoel ik: de procedure bepaalt uiteindelijk de line target

JJ

-----

Inheritance: Extension,

2009-08

If you can implement procedures for Get, you should be able

to override the default implementation, and also ditch the

storage variable too.

JJ

-----

System Command Extension,

2004

The Object, Type, Interface and State’s Gets and Sets can be redefined by filling the procedures with code.

> 2009-08-31: So show the implementation right inside the system command definition, instead of seeing it in the extension ‘event’ notation?

JJ

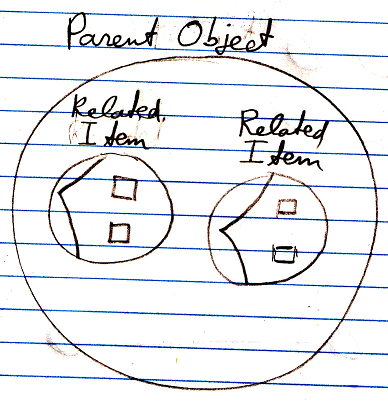
### Extend Connectors and Connections

The Connectors and the Connections articles have to be extended, because they are supposed to express every system command call as a connector or connection, but they don’t.

### System Objects

#### Parent contains references to sub-objects

A relation to another object is part of the parent object. A parent object is a real object. Its related item is not a real object, but a *pointer* to an object. The system interface of a related item determines which object the parent will refer to.



The pointers, so the system objects, are part of the *parent* object. Objects referred to by a parent object are not part of the parent object, but the *references* to the object *are* part of the parent object. Objects such as a related list or a related item are system objects that are inherently part of the parent object. Only the objects referred to are not part of the parent.

The related item system object is bound to the *parent* of the related item, not bound to the object itself. The parent object determines the target of related items, through the commands of the system object.

A *related item* represents an object in relation to an another object.

#### Pointer-to-Pointer

Pointing to another parent’s related item, means that the *other* parent has control over the eventual target pointed at. This is exactly what a pointer to pointer is applied for: giving another parent control over the eventual target pointed at.

Synonyms for pointers to pointers are:

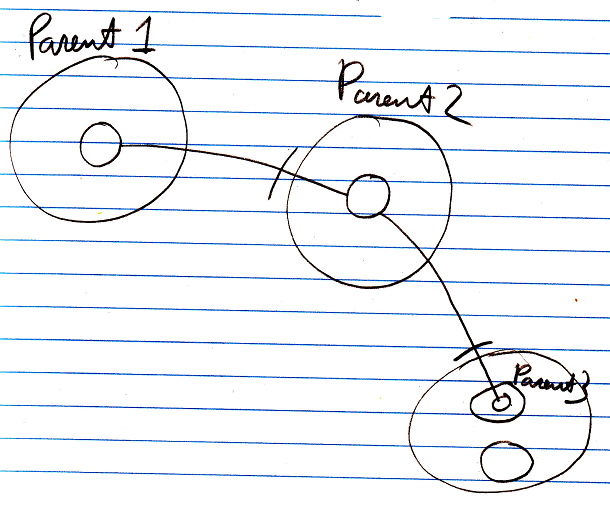
- Pointer to a related item

- Related item pointing to another related item

- Reference to an object reference

#### Multiple redirection

When a related item points to a related item, that related item can again point to a related item, and so on.



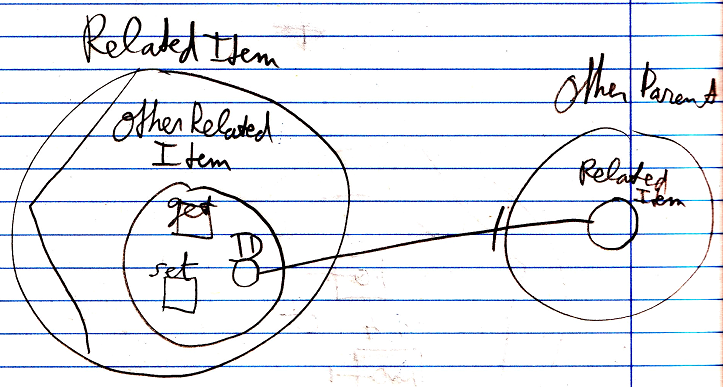
That’s how you create multiple pointer redirections.

So Parent 1 delegates its sub-object to Parent 2, granting Parent 2 control over the eventual target pointed at, which Parent 2 does, by deciding the eventual target is a related item inside Parent 3.

Parent 1 gives Parent 2 control the eventual target pointed at.

#### Other Related Item

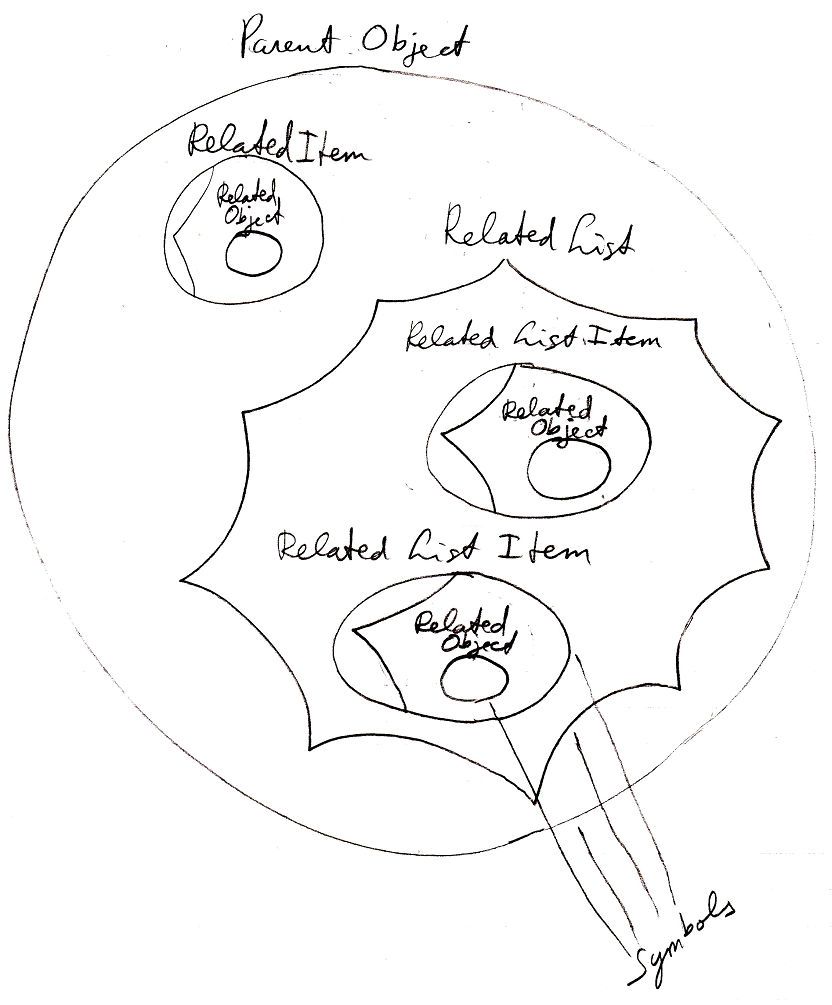
Because the contents of the target object are probably not visible in the diagram, when the system interface is opened, the target of the related item will be connected to the Other Related Item . ID or Other Related List Item . ID of the system interface:



#### System Objects

A *related list item* is a system object. A related list item is contained by a list, that is contained by a parent object.

< Nice picture: >



-----

Related Lists,

2008-08-10

You can’t always display all items in the list, if there are too many. You have to have a way to scroll through them. And parhaps you should be able to open up a table view.

JJ

### Qualifiers

Vraag 1:

Waar en hoe worden line targets opgeslagen?

* In een Line Target object <geen goede naam>. Deze klasse bevat een Entrances Integer Array. De Entrances Integer Array bevat de ordinalen van de symbolen to enter. Een ordinaal is het symboolnummer binnen zijn parent.
* Elke line wordt zo opgeslagen binnen het object dat de line bevat.
* De line source hoeft niets te weten van zijn omgeving en de inhoud van children of cousins om er toch naar te verwijzen
* Pas als de echte line crossings (hoe heten deze symbolen die gecrossed worden?) moeten worden achterhaald, dan worden er traces uitgevoerd. Dit gebeurt bij opvragen van informatie over een symbool.
* Dit heeft tot gevolg dat bij veranderingen in het systeem, de line target naar onbedoelde symbolen verwijst. Daarom worden bij bepaalde systeemveranderingen (optioneel?) de line sources opgeheven (of tijdelijk onschadelijk gemaakt?) < moet de term *line source* niet ergens geïntroduceerd worden? >

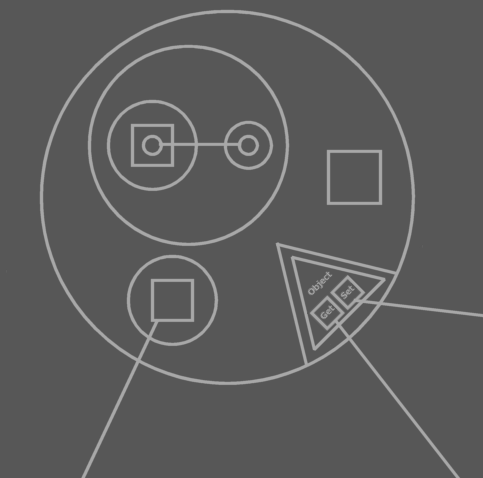
Dus krijgt elk zichtbaar symbool toch een nummer? Ondanks dat systematiek opgeslagen is in de object klasse van de module klasse?

### System Interface Notation

System Objects,

2009-08-31

Notation of the system interface *can* go together with the regular contents of the symbol.



JJ

### Code Base Objects

Because procedures have system procedures themselves, you’d think that an system procedure also has an system procedure. That one isn’t true. System procedures don’t have system procedures themselves.

-----

System Objects

2004,

If you see the system interface as an object, though, you can regard the system interface to have the same type for every object symbol. Procedure symbols would have a different system type. All symbols would derive from this system type then.

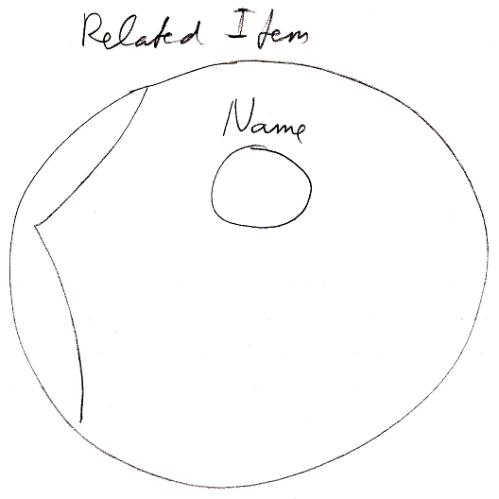
JJ

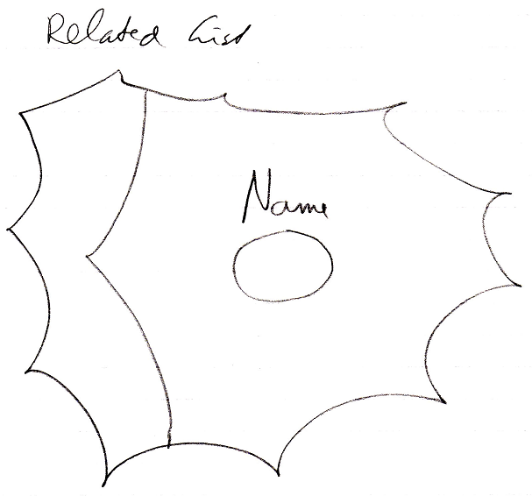
### Preliminariness of System Interface Notation

aspect as sub-object implementation.

Maybe mention the possibility of implementing some aspects as sub-objects, using the Name aspect as an example…

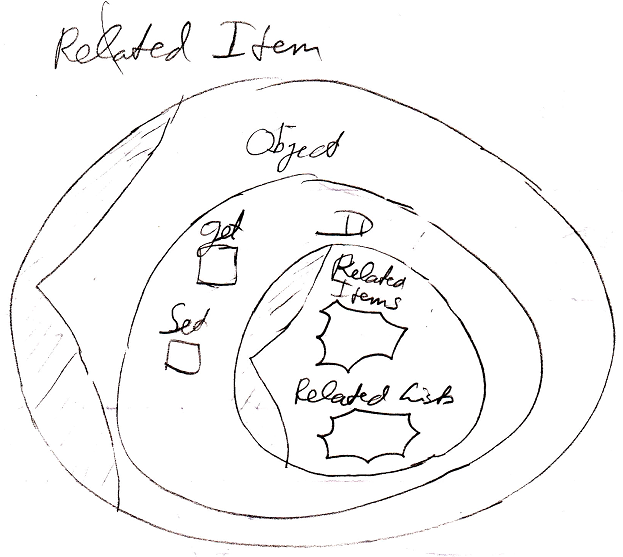
The diagrams below show a Name member inside the system interface of a related item and a related list.





(also show, that the name property has a system interface itself, with a **Get** and a **Set** command. )

Other possibility:



### Form

- A lot of the times it is interesting to introduce the diagram almost immediately. And most of the times the diagram is quite self-explanatory and does not require sentences, such as: "The system member Add is part of the system interface of a related list and is displayed in a diagram as follows:"

-----

- Perhaps do not introduce the complete system interface of a system object. am not going to explain it in the article of the system object, so why bother reader with it. Give a conceptual explanation without the whole system interface. If you must show the whole system interface, show it in a separate article, after all system objects and system aspects have been explained. Call it for instance: 'Related List Item System Interface'.

-----

- In each article it seems to be repeated, that you can extend the system commands with extra functionality. Why not leave that out, and just mention that in the article, that actually explains extending the system interface. Do not make so many references to other issues... that will make the articles more complicated.

-----

- Do not say all the time: 'related item or related list item'. Just mention somewhere that a lot of times when saying 'related item' it also means, that it applies to a related list item and it can be determined with common sense which things.

-----

- Get rid of the separate Text Code folders

-----

- Move explicit notations further down below

### Phrases

sometimes there are multiple variations for system commands in order to separately access control them for different situations.

-----

A system object is represented as follows: <Bluntly a simple display of the system interface notation> It is like breaking open another object, showing its inner workings by showing the system object, that it is based on.

-----

System Objects,

2004

The combination of a Get and a Set as such is called an *assignment*.

JJ

### Naming

Why do I call them Object Get and Object Set? Is that proper English...? Why don't I call then Get Object and Set Object?

Probably because I want to mention the aspect first? You know what? The naming does not matter that much. The concepts do. A PROBLEM IS, THAT IN THE SYSTEM INTERFACE ARTICLES

I WILL GROUP COMMANDS BY ASPECT, SO THEN THE IDENTIFIERS WILL LOOK DIFFERENT. I AM GOING TO HAVE TO SYNCHRONIZE THAT TO THE 'OLDER' ARTICLES THEN...

- I misspelled Existence. I spelled it as Existance.

- Rename the Execute aspect to the Execution aspect.

-----

System Objects

Object of class zou aan een gebruiker ook kunnen worden uitgelegd als 'context'

JJ

### Working methods

- The System Commands articles may be further extended with details and more exact descriptions, possibly coming out of other material.

### Details

System Objects,

2004

There are two things you can do with a value: copy it or transform it.

JJ

-----

System Objects,

2009-08-31

There are no assignments between the Value and Object aspects… cover that somewhere?

JJ

-----

Conversion,

2004

If the state isn’t assigned to the same type then a conversion must take place.

JJ

-----

New Computer Language,

2009-09-03

What about a two-dimensional array? How would that look in the new computer language?

JJ

-----

New Computer Language,

2009-09-03

Nice phrase:

2004: Usually you won’t work with memory management. You’ll simply use for instance Integer or Array objects from the Math and Data modules. Those objects manage memory.

JJ

#### Command with Return Value versus Object with Get Command

< The exchangeability of method and object notation >

< You CAN display a method as an object. You’d then display it as its return value(s) >

-----

You can view a procedure as the object that it returns.

-----

< Picture: Procedure with some private contents and one public object with some contents >

=

< Picture: Object returned >

-----

Actually, the objects symbols I’ve introduced before *are* the object representation of procedures. They are representations of procedures that return objects.

Here you see that you have multiple representations of something: you can see the system as a bunch of procedures or you can see it as a bunch of objects, but it’s still the same system.

If you regard something a property, which you often will, the primary representation of it is its object representation. If you regard something a method or if the procedure doesn’t return a value, then its primary representation is the procedure representation.

Conversely, a procedure can apart from it’s usual representation, be represented as the object that it returns.

> 2009-06-28: Then you have to just make it an object symbol and implement the object get system procedure.

#### Object = Collection?

System Objects,

2009-04-08

I want to add this idea to to the System Objects documentation:

An object is a collection of other objects. So in a way an object is always a collection. Yet, it is not represented by a nonagon shape, that represents multiplicity.

You can have a nonagon be a collection of objects of any arbitrary class.

What then defends the use nonagon and what defends the use of an object, because both are really collections.

I have to think that over once.

JJ

#### Object Target Reference = Value??

System Objects,

2009-08-31

Internally, an aspect such as Object is stored as the ID of the object, which is a Value, stored as a Value aspect of the ID object.

JJ

#### Design Time = Run Time

System Objects / Design Time = Run Time,

2009-08-31

Typical design type aspects:

- Ancestry

- ‘Shape’

JJ

### Assignment Ideas

Assignment,

2008-07-03

Een reference to reference maakt een object assignment zeg maar statisch???

En andersom: een object assignment kan worden vervangen door een verwijzing naar een verwijzing van een object?

En een value assignment dan? Kan dat worden vervangen door een verwijzing naar een verwijzing? Nee de gekopieerde waarde leeft daarna onafhankelijk voort.

Een class assignment is ook een reference to reference.

Hoe kunnen deze twee dingen zo inwisselbaar zijn?

Iets statisch (een reference to reference) en een actie (een object assignment).

JJ

-----

Computer Language Coding Principles,

2008-07-04

An object assignment, and a pointer to a pointer are not completely interchangeable, but it is close to it.

JJ

-----

Assignment,

Pointers to pointers as a concept,

2008-08

A related item can point to a 🡪1 related item contained by another parent or to a 🡪n related list item contained by another parent. Redirecting to another related item or related list item may be accomplished by extending the object get and object set procedures of the related item. The extension procedure basically constitutes, or instantiates, the redirection. So a pointer to a pointer would be accomplished by procedure extension, instead of implementing it in the code base. However, this will not work for related *list* items, because all items in a related list have the same decoration, and redirecting a list item to an arbitrary other related item or related list item would require each list item to have its own decoration.

< If you make a related item return an object out of another parent, then this decoration with procedures is functionally the same as making the related item a pointer to related item in another parent. But this implementation of a pointer to a pointer will not work for related list items, because list items do not have their own individual decoration with procedures, so that’s why pointers to pointers still need to be part of the code base. >

JJ

#### Integrate these Sections

##### Value Direction

Read-write direction is something so basic in computer language, that it has a separate line type. Value direction is indicated by a *wavy line*.

A value connection is actually not a real connection between symbols, but a value assignment.

A wavy line is actually a good symbol for it, because it looks like a less direct connection than the other line types. The wavy line looks like a less direct connection, because the complete length of the line, curves and everything, is much longer than a straight line. A wavy line also indicates *heat*, or activity, more than the other line types. Value direction is an *action*.

Read-write direction, input-output direction and value direction are synonymous terms.

A read of an output parameter is always accompanied by a write to another object. A write to an input parameter is always preceded by a read from another object. A read operation is always accompanied by a write operation. A value is read from one object, only to be written to another object.

> Actually, writing is not the only purpose a read can have. You can also read for access, apart from read for assignment, but this will be thought through later.

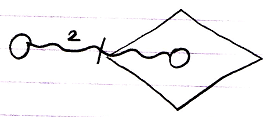
##### Clone

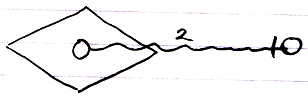
For more complex objects, value direction can mean an object will be cloned up until a certain cloning depth. When the values of an object are cloned up until a certain depth, no references to original objects will be taken over by the clone. Clones like that never have references to existing objects, but always contain entirely new objects or object references that are Nothing.

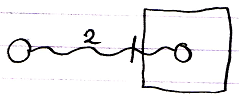
A single-value transfer is actually the equivalent of a cloning operation with a depth of 1.

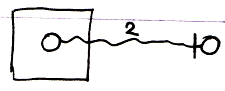
There are other types of cloning in which clones do end up with reference to existing objects, but those types of cloning will be covered in the article Clone in the Editing Concepts.

These are examples diagram expressions of Value cloning operations:







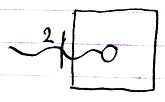


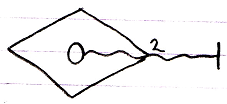
The number 2 stands for the cloning depth.

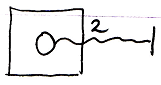
The access mark indicates value direction.

A single-value transfer will be the same as a cloning depth of 1, but the number 1 will not be shown then.

Here are examples of expression of value cloning access connectors:





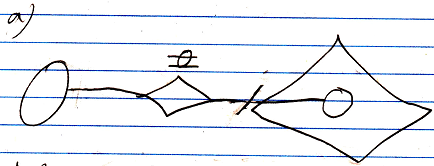


#### More Ideas

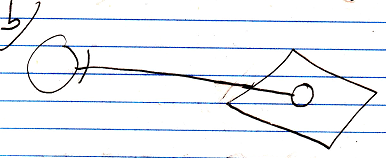
Assignment,

2008-08

A reference parameter is assigned an object.



The assignment of an object can also be replaced with the *redirection* of an object.



-----

Parameter passing in text code with an object assignment call in it:

A ( O o= B ( ) . O )

Command A is called, and its Reference In parameter O is assigned the output O of the command B. So no temporary variable is needed to first pick up the output of B and then assign it as input of command A.

JJ

#### From the original Symbol documentation

##### Assignment

Before I explain using arguments, I must introduce another basic concept: *assignment*. I have to explain it, because to use input arguments you have to *assign* a values to it. Likewise, to retain output arguments, you must *assign* the output values to other objects.

*An assignment*:

A = B

consists of two actions: a **Get** and a **Set**. First you **Get** **B.** Then you **Set A**. An assignment always consists of one **Get**, immediately followed by one **Set**.

There are two main types of assignment: *state assignment* and *line assignment*.

State assignment is the most basic, and the easiest to understand. The values of one object are copied to another.

A = B

The value of **B** is copied to **A**. The state assignment does not make **A** and **B** the same object. They are two *separate* objects that contain equal *values*. Change the value of one object and the two objects differ again.

To make **A** and **B** represent the same *object*, you have to do a *line assignment*:

A  B

***Pronounce: A’s object is B***

***It’s an equals sign with the letter o in it.***

This is an *object* line assignment. The object line of **A** is set to symbol **B**. The result is that **A** points to **B**.

The operator **=** is not reserved for state assignment. Instead, the operator for state assignment is commonly: ****.

A simple diagram notation of an assignment is as follows:

|  |  |
| --- | --- |
| State assigment | Object assignment |

Or:

|  |  |
| --- | --- |
| State assigment | Object assignment |

The result of object assignment:



is:



But you can also view it the other way around. The following:



Is the result of an object assignment:



This is an important way of viewing it, because most of the times you’ll only see the result of an assignment and not the assignment operation itself. The assignment operation *is there*, however. Therefore an even simpler notation of assignment is not to show the assignment at all.

Unfortunately, the result of *state* assignment is not visible in diagram code as a line, so you have to display the explicit assignment.



A shorthand, quick notation of an assignment is the following:

|  |  |  |
| --- | --- | --- |
|  | = |  |

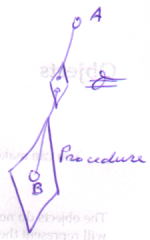
There are more ways to draw out an assignment, but the ones here are by far the simplest and most easy to understand. Other notations are covered in another chapter.

State assignment is the most basic assignment. Line assignment is based on it. A line target is stored in an **Integer** object. So when assigning the line target, you’re actually assigning the **State** of this **Integer** object. Drawing a line assignment as a state assignment is a more difficult, but useful notation. However, in this chapter I will stick to the simpler notations.

You can work with the *type* and *interface* lines the same way: there’s also type assignment and interface assignment. Likewise, you can work with line assignments for *procedure symbols*. However, I’ll mostly talk about state and object assignment.

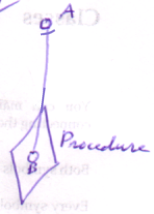
##### Distinguishing Assignment Source and Target

There’s one more ambiguity in argument access that I will resolve here. In an assignment call in diagram as follows:



there isn’t anything that tells which is the source of the assignment and which is the target: is it **Procedure . B**  **A** or is it **A**  **Procedure . B** ? You could name the arguments of the  (*object is*) call: name them **Get** and **Set** to make out the difference, but usually another method is adopted.

Assuming the assignment is really **Procedure . B**  **A**, the effect of the assignment is:

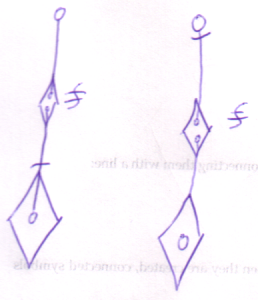


To show the ‘direction’ of the assignment, the access symbol of the resulting line is shown in the assignment call:



In this case, the access symbol is about the *effect* of the assignment, not about the direction of ’s **Set** argument.

In the case of state assignment, the result is not a line, so logically the state assignment call can’t ‘borrow’ the access symbol from the resultant line. However, in line assignment, the target of the line, the source of the assignment (!), gets the access symbol. Conversely, the source of the state assignment gets an access symbol:



The rule is: the source of the assignment gets the access symbol. An oddity is that the *source* of a line assignment is the *target* of the resultant line.

Direction can be determined by assumption of the most common situation. Therefore, the access symbol can be left out it the resultant line is outwards.

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

For state assignments it’s advised to always show the access symbol.

##### Executions & Parameters

###### Brainstorm

Summary of Old Subdivision

* Basics of assignment
  + Line assignment
  + State assignment
  + The notation I'll use here
* State assignment is the basic assignment

New Subdivision

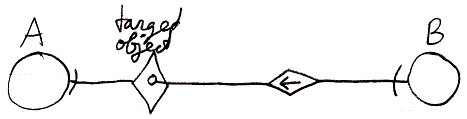
* Assignment
  + The terms Get and Set, introduce them
  + A Get and Set combine to an assignment
  + Line assignment
  + State assignment
  + Use the simple notation
  + ~ State assignment is the basic assignment

### Other Ideas

Alternative Assignments in a Diagram,

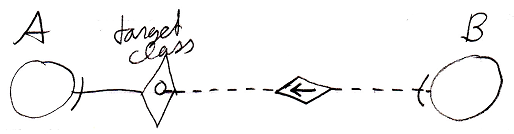
2008-07-26

One of the prime examples of an alternative assignment, compared to normal assignment and object reference assignment, is to assign the final target of object redirections of one symbol as the object of another symbol. This would be implemented as first calling the Target Object system command and then assigning its return value as the object of the other symbol. Below is the expression of that in a diagram.



Calls to system commands are explained in the article *Calling a System Command*.

Another example of an alternative assignment, is to assign the final target of class redirections of one symbol as the class of another symbol. This is implemented as the a call to the Target Class system command and then assigning its return value as the class of the other symbol. Below is the expression of this in a diagram.



JJ

-----

Alternative Assignments in Text Code,

2008-07-26

The concept of alternative assignments was already explained in the article *Alternative Assignments*. The current article displays the text code of some prime examples of alternative assignments.

One of the prime examples of an alternative assignment, compared to normal assignment and object reference assignment, is to assign the final target of object redirections of one symbol as the object of another symbol. This would be implemented as first calling the Target Object system command and then assigning its return value as the object of the other symbol. Below is the expression of that in text code.

B o=🡪 A . Target Object

Another example of an alternative assignment, is to assign the final target of class redirections of one symbol as the class of another symbol. This is implemented as the a call to the Target Class system command and then assigning its return value as the class of the other symbol. Below is the expression of this in text code.

B c=🡪 A . Target Class

JJ

-----

System Objects,

2009-08

- Related list item may support the same interface as a related item, as an aid in concepts that apply to both related items and related list items.

- I do not like the fact, that you do not get to reference a list. But that is a detail.

- Also, the Class of a List can be fixed. This is not mentioned anywhere yet.

JJ

-----

System Objects,

2009-09

- More system commands

- Make sure you include all system commands thought of later.

- This also extends the matrixes in other articles, that show a bunch of different forms of something.

- Pointer to Pointer

- Also cover pointer to pointer notation for system commands and access connectors

- I am not sure that I want it to be implied by an inward reference... I insist on the arrow notation, because it is very special behavior and I want it to stand out.

- Notation of Object Class and Reference Class are not unambiguous yet.

- Active connections cause a Get. For commands, it also means, that an access requires the command to have run? Hmm... an explicit execution? Perhaps not. Perhaps you just should not see that implication.

JJ

-----

System Objects,

2008-08-08,

Toegang van het ene object naar het andere met puntnotatie,

is ook een concept. Je moet namelijk access operators gaan definiëren, en toegang als aparte actie zien, ook om die actie apart te kunnen access controllen.

JJ

-----

System Objects,

2009-09-06

Also clearly state, that everything comes down to system command calls, even when you are accessing objects.

JJ

-----

System Objects,

2009-08

< Om iets alleen te mogen gebruiken als class kun je de access connector line gestreept tekenen (en ook de shape). Als je iets wel als object mag gebruiken, alleen het wordt alleen *gebruikt* als class kun je de *shape* met gestreepte lijn tekenen maar de access connector (inclusief line) ononderbroken laten. >

JJ

-----

System Objects,

2009-08

There may be a difference between the fundamental notations of system commands that do return something and system commands that do not return anything.

JJ

-----

Connections, Connectors,

2009-07-09

If you can not display an access mark for outward directions, why can you display an access mark on a set access connector?

JJ

-----

System Objects,

2010-05

I have the idea, that you could distinguish accessing the reference from accessing the object though use of the access mark, because that indicates an active get, an active access. But then the notation may come in conflict with other simplified system objects notations.  
Or maybe in that case of those simplified system object notations this is alright, and in any other case accessing a reference or accessing an object is distinguished by the presence or absence of an access mark.

JJ

-----

System Objects,

2010-05-12

The Name aspect is missing from the command call notations,

and probably also other aspects.

JJ

### Enumeration of Attributes

A symbol might simply become an attribute depending on its characteristics. Attributes might be spread around an object’s list of related items. (See Attributes in the Objects chapter.)

The\* attributes could be returned by the\* object as a separate\* list, instead of looking them up in an object’s collection of related objects.

An object might return a list of just its attributes, instead of just all related objects.