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

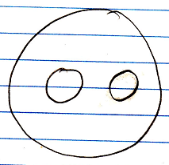
## System Objects Misc Issues

### System Command Calls by User

A user will often execute Gets and Sets and other system commands when connecting symbols together, but they will only see the connectors and the *result* of a Get and Set, but never the explicit Get and Set calls. The system commands are executed as the user builds up a diagram.

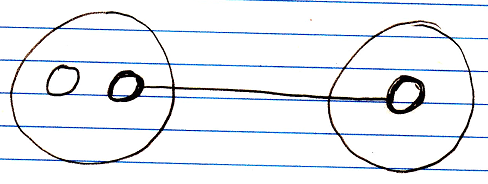
### Objects Floating Around

Objects are never directly accessed. They are always floating around somewhere you cannot touch. You are always accessing an object through an *object reference*. You are always dealing with *references* to objects, never with the object directly.



The smaller, contained circles are *references* to objects, even though the bigger circle seems to be the sole container of the objects themselves. An object does not really contain sub-objects. An object contains pointers to its sub-objects. Even when the object seems the sole container of the other objects, the other objects are really only referenced. You do not see the actual object. You are only seeing references to it.

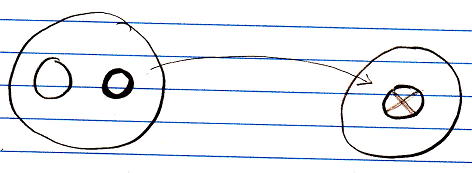
Another symbol can start referring to the same object, making the object all of a sudden not part of a unique container anymore.

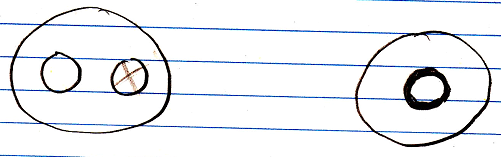


When you annul the object reference in the original parent, the second parent all of a sudden becomes the sole container of the object.



The object has moved from one parent to another.





Objects are always just floating around like that. They do not have to stay in a fixed spot.

In reality the objects do not move at all. They are physically stored in the same spot all the time, no matter where they appear to be. An object can just be freely referenced from anywhere, because objects are always accessed through references.

Even when you *create* an object, you are not directly in touch with the object. The object is immediately assigned to an object reference. Also: when you assign a *value* to an object, you do not assign the value directly to the object, but you assign it through an object reference.

Each object reference gets its own identifier, even when an object reference is Nothing. An object itself, does not have an identifier. An object can be given a Name attribute, though.

### System Command Extension

One thing that makes it important to be aware of system commands at all is *extension* of system commands.

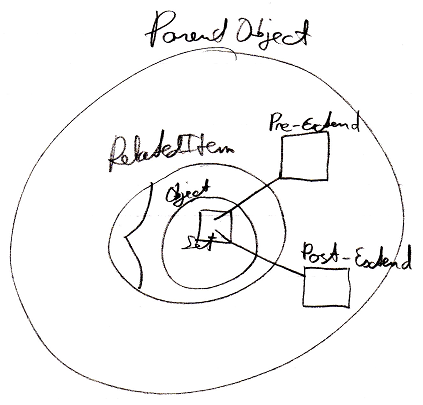
A system command, such as Value Get, can be decorated with a procedure, that determines the value that is returned. It is the new computer language’s equivalent of Getters and Setters or Property procedures. The Object Get command can be extended as well, which makes you able to further control the connection between one object and the other.

Any system command can be decorated with a Pre‑Extension, Post‑Extension or Override.

System command extensions are implemented the same way as ‘normal’ command extensions. Command extension is put under the topic of *Inheritance*, because it is an extension technique.

System command extension will get a notation, that complies to the notation of normal command extension. The code base will implement system command extension like normal command extension and the way extension is implemented inside the code base will influence how the eventual notation will look.

It might look something like this:



But the exact way in which command extension is implemented will be covered by the *Inheritance* articles.

### Parameters For Objects

**[Preliminary documentation]**

#### Concept

This is a preliminary description of the concept. The details are still to be worked out. It could be that in practice, when the new computer language is up and running, the details that have to be solved, will come to light straight away.

In other programming languages there are *getters* and *setters*, which are also called *properties*. Those are replaced by the fact, that any related object has a set of system commands, such as Object Get and Object Set, that can be extended with extra code around the Get and Set actions.

For properties it is sometimes handy to hand a *parameter* to the retrieval of a value, and then a certain value is returned. For instance to return the pressure value of a sound wave at a certain time you could have a Pressure object, that returns a value when you pass the Time to it as an argument.

Pressure ( Time )

Even through the retrieval of pressure could be made a command with a parameter, one might want to see Pressure as an *object*, rather than a *command*.

You can use a related object for that. A related object has a system interface, that allows you to let the eventual object it displays be determined by a procedure. The system interface controls what is returned as the related object. You can extend the Value Get and Value Set system commands. The new computer language must allow you to be able to add extra parameters to system commands, or add extra sub-objects to a system aspect, such as the Value aspect, creating a single parameter for both Value Get and Value Set at the same time, and the new computer language should also allow you to add sub-objects to the whole system interface, to give the retrieval and assignment of any aspect the same parameter. So it is not really the object itself, that gets a parameter, but the related object, that gets a parameter. An object *reference* gets a parameter. That is why the parameter needs to be part of the system interface.

Because you can add a parameter to the whole system interface which extends every system command with a parameter, the new computer language should supply the capability to select which system commands actually get extended with the same parameter, and whether the Time parameter is required or optional. Time is a sub-object of the related object’s system interface, so it is not really a parameter of a command. However, it does extend the system commands with a Time parameter, so a sub-object of a system interface is always called a parameter, but it is called a parameter of a *related* object instead of a parameter of a command.

Adding parameters to the system interface of a related object or extensive extension of system commands is a way to let a retrieval procedure be represented by an object instead of a command.

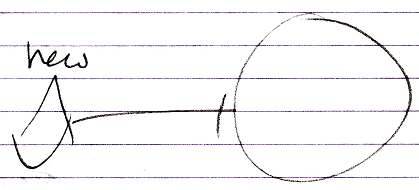
A command in the new computer language can have multiple return values, but when you convert the command into the retrieval procedure of a related object, the command will actually have a single return value. So in this case, you do have the concept of having only one single return value, unlike commands, in which you can have multiple return values.

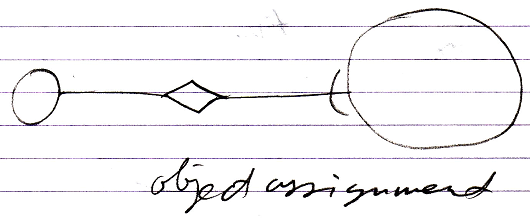
There are no plans yet to make a command, that is a retrieval procedure, and a related object with an extensive retrieval procedure, two completely equally present views on the same thing (like other flat and structured interchange concepts within the language, like exchangeability of class commands and command parameters).

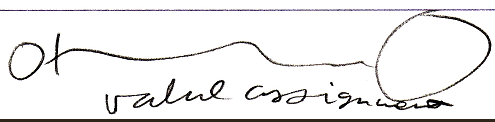
A query is also an example of a related object or related list with an extended system interface, that determines the item, list or result set eventually returned. Dependent on the parameters of the related object, the outcome is calculated.

#### Diagram Notation

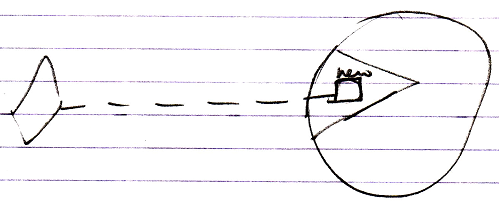
Default system commands can be called with an easy notation, that does not show the system command definition:



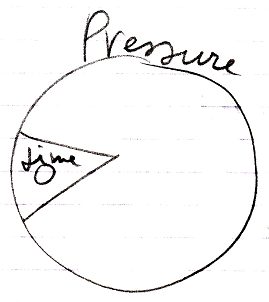




Even though you can also display the system interface and show a call to the command definition:



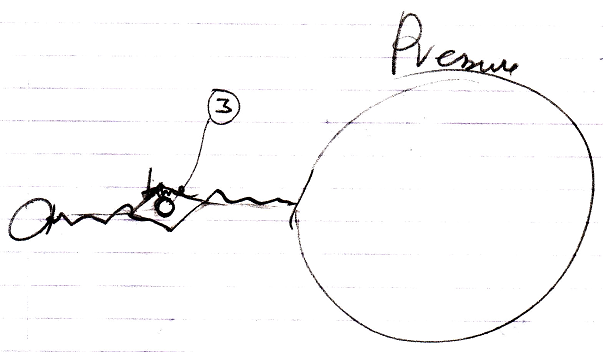
When you very much customize the system interface, you do not always have a standard notation for a consult of the system interface anymore. Giving a related object’s Object Get and Object Set a parameter, you have to display the system interface all the time.



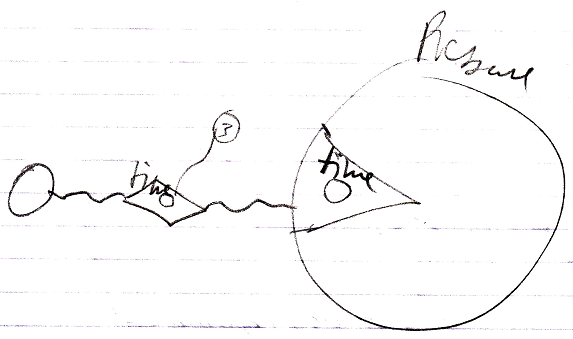
This shows, that the related Pressure object has a Time parameter.

In this case the whole system interface is extended with a parameter, because the Time parameter is not shown in a specific system command or specific system aspect, but shown inside the whole system interface. This means, that with any system command you can supply the Time parameter.

A call to the system command, such as value assignment, will show the Time parameter:



The notation above might not the best one. You may want to show the Time parameter in the related object’s system interface at all times:



This clearly depicts, that the Pressure related object has a Time parameter. You can not go around this parameter.

### Referrers

#### Concept

Objects can have references to other objects. A referenced object may not aware of its referrers, but it might be an option to explore for an object to have all its referrers registered in a list.

The\* referrers are not\* the\* parents containing the\* references to the\* object, but\* the\* referrers are the\* *references* to the\* object themselves.

When\* a related item is set to point to a certain object, the\* **Related Item . Object . Set** command will update the\* target’s list of **Referrers**. So the\* *referrers* update the\* target’s **Referrers** list. The\* referenced object does not\* update the\* **Referrers** list itself.

The\* **Referrers** list consists of references *back* to the\* referrers, but\* that does *not\** mean the\* object in turn becomes a referrer of the\* referrer again.

An object can\* have a referrers list, but\* an object reference, so a related item or related list item (see the\* *System Objects* articles), can\* also have its own referrers list for references that refer to references.

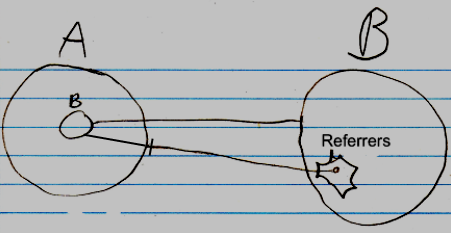
##### Not\* Supporting the\* Referrers Concept

An object could\* choose not\* to support the\* **Referrers** concept, if\* the\* programmer knows, that this object will be referenced so many times, and there is so little interest in knowing all its referrers, that it would\* be ridiculous maintain a list.

But\* by default, the\* **Referrers** concept is always supported.

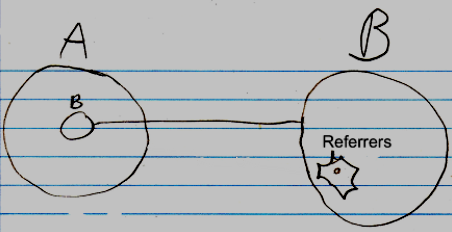
#### Diagram Notation

The\* referrers of an object are simply displayed as a sub-list called **Referrers**, every item of which points back to the\* references to the\* object:



The\* entry in the\* **Referrers** list is pointing to a related item in the\* parent object **A**, not\* directly to an object.

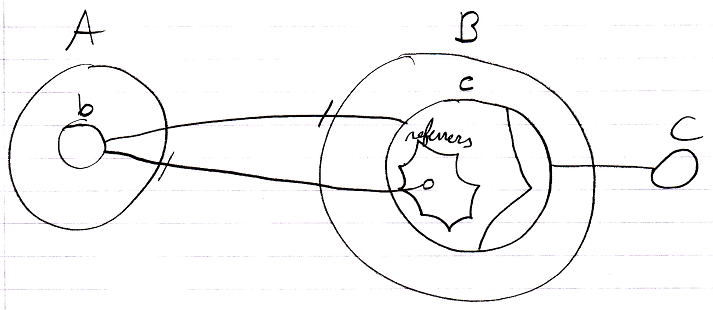
The\* lines coming out of the\* referrers list are usually not\* shown, because\* a line tied *to* an object already *implies* a referrer. The\* diagrams will have more features later, and the\* referrer lines would\* obscure the\* picture.

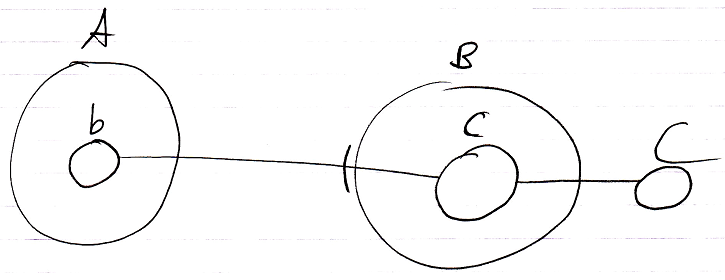


Even the\* whole referrers list may even be left out of the\* diagram by default, but\* it is not\* clear yet, if\* that is the\* way to go.

If\* something refers to a reference, then\* this may look like this in a diagram:

**b** in **A** is a reference to the\* reference to **c** inside **B**. To display the\* referrers in the\* diagram, you\* could\* <>do something like this<>:





### Class Referrers

#### Concept

The\* *Referrers* article explained how an object can\* be made aware of its referrers. A *class* can\* also be made aware of the\* objects using it as a class.

Classes are implemented as an aspect. That concept adds an object reference to the\* system interface. This object reference points out which other object is its class. So oddly, an object reference, that points out the\* class, is already added to the\* class’s list of referrers. The\* classes are registered inside the\* same list of referrers as object referrers. This is actually just fine. The\* **Referrers** list is supposed to be a low-level view on the\* referrers.

A class is usually only *used* as a class, and not\* also used as an object, so in practice, the\* **Referrers** list of a class, actually already *is* a list of class referrers. So a separate list of **Class Referrers** will not\* be implemented.

But\* if\* in the\* future there is a need to also maintain a separate list of class referrers, a separate **Class Referrers** concept could\* be implemented. In that case, when\* a related item’s *class* is set, the\* **Related Item . Class . Set** will update the\* target’s list of **Class Referrers**.

##### Not\* Registering Class Referrers

The\* amount of referrers of a **Number** *object* may be small, but\* the\* amount of referrers of the\* **Number** *class* is humungous. The\* class will even have a **Referrers** list, when\* the\* class is not\* a created object, because\* **Referrers** applies to both symbols and objects.

You\* would\* want to turn the\* **Referrers** concept *off* for the\* **Number** class and *on* for **Number** objects. But\* the\* problem here is, that a class is a blueprint for an object. An object only supports **Referrers**, because\* the\* *class* supports it.

The\* first solution proposed was to simply not\* support the\* **Referrers** concept for classes that are widely used. But\* then\* for widely used classes, the\* **Referrers** concept never be supported. That is against the\* idea of supporting the\* **Referrers** concept by default.

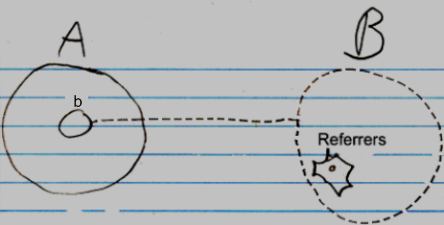
If\* you\* can\* not\* stop a class from supporting **Referrers** without stopping objects from supporting **Referrers** at the\* same time, then\* the\* **Referrers** concept will not\* be widely used anymore.

Therefore, you\* are going to have to specify for a symbol or object, that it is a non-practitioner of a concept. Derivation of objects will take over the\* specified concept, but\* not\* the\* non-practitioner aspect. Or perhaps instead of calling it non-practitioner, you\* could\* call it **Objects Support Concept Referrers**, or something.

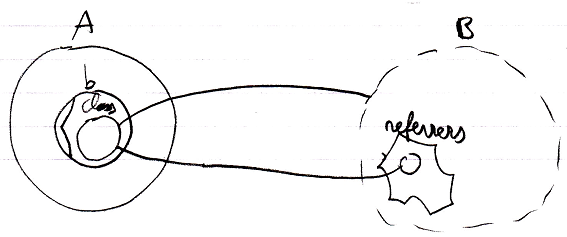
#### Diagram Notation

< The\* notation of a reference to an object reference’s class needs to be determined in the\* future. >

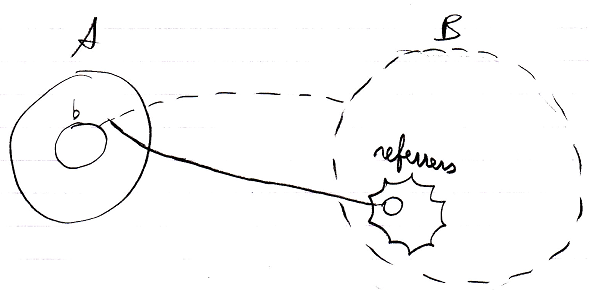
Because\* the\* concept of referrers simply also functions as the\* concept of class referrers, it can\* be displayed in a diagram the\* same way, except, that classes and class references are displayed with dashed lines.



The\* reference line of the\* item in the\* **Referrers** list is displayed, then\* it has to point to the\* class redirection of symbol **b**. There is no final notation yet for a to something else’s class. But\* a preliminary notation could\* either be a reference to the\* **Class** inside **b**’s system interface:



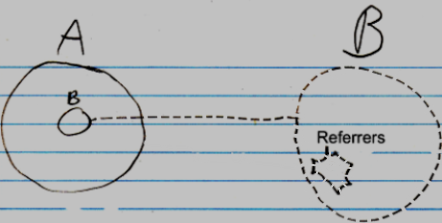
Or a reference line connected to **b**’s class line:



The\* referrers are pointed at by solid lines, because\* they are just references to the\* objects, that use it as a class. No implicit notation of making the\* referrer lines *dashed* will be used here, because\* that will introduce too much ambiguity in the\* diagram notation.

As mentioned in the\* article *Referrers*, it is not\* clear yet under which circumstances the\* whole referrers list might be completely left out of the\* diagram.

If\* a class defines that its objects support **Referrers**, but\* the\* class itself won’t register its **Referrers**, then\* the\* Referrers list of the\* class will be drawn out with dashed lines.



Obviously, the\* inactive referrers list will not\* contain any object references.

### Referrers Versus Related Objects

Referrers are handy, when\* so many classes relate to another class, that the\* other class does not\* want to maintain a separate list for each class that links to it.

It is also handy for when\* a class can\*'t be aware of its related classes, so can\* not\* automatically get a relation back to classes, that want to link to it. In that case the\* other class can\* not\* establish a dual relation with the\* remote class, probably, because\* it does not\* have permission to alter the\* remote class. Or the\* remote class denies dual relationships to it altogether.

To make the\* remote class or object aware of its referrers anyway, you\* can\* let it support the\* referrers concept.