|  |
| --- |
| Circle Language Spec: Relations |

## Relation Synchronization

In a dual relation between classes, one class relates to another and the other class relates back to the first class.

But that is not enough to establish a full relation.

An object of one class can refer to an arbitrary object of another class, which then refers back to an arbitrary object of the first class again, but not necessarily the object we started with.

To see to it one object relating to another always makes the other object relate back to the first object again, you have to keep the two counterparts of the relation synchronized. When you make a change to a sub-object’s target, the other side of the relation also needs to be updated. This is called *relation synchronization*. Relation synchronization makes something between two objects melt together to one relation. Only when the two unary relations are synchronized, then it is a full dual relation between classes.

If two unary relations are not synchronized, then they are just two separate unary relations, that have nothing to do with eachother.

A complete dual relation consists of three parts:

- one class has a sub-object of another class

- the other class has a sub-object of the first class

- the two unary relations are synchronized

When you create a relation, you are adding those three elements. In the new computer language you are still able to create just a unary relation. In a programming environment you should be able to automatically turn a unary relation to a dual relation. In a programming environment you should be able to turn two already defined unary relations into a single dual relation.

Relation synchronization is not present in the class structure. Relation synchronization happens between *objects*.

As explained in the article *Relations Between Objects*, any object, that as a relation to another object, gives the other object one relation back to the first object. In other words: one reference inside an object is tied to one reference inside another object.

It doesn’t matter if any of the two object references resides in a list or not. When you let one of the object references link to another object, then the link from the old counterpart to the object should be broken and a link from the new counterpart to the object should be established. The object synchronizes the old counterpart *out*, and it synchronizes the new counterpart *in*.

Relation synchronization means ensuring the integrity between the two counterparts of a relation. When you assign a Lid to a Jar, the Jar is also assigned to the Lid. So whenever a Jar changes its Lid, the original Lid’s reference to the Jar is annulled, and the new Lid get a reference to its new Jar. This actually explains synchronization between 1 🡪 1 related objects. Synchronization happens when assigning a related item. When assigning a related item, the related item gets a reference back to the first item.

Relation synchronization happens when you assign a related object. When you assign Lid . Jar, then synchronizing the relationship, consists of, in turn, assigning Jar . Lid.

For every relation type it works in a different way.

There are three relation types:

1 🡨🡪 1

1 🡨🡪 n

n 🡨🡪 n

But synchronization is managed separately for each end of the relation, so synchronization is managed in the following four of ways:

1 🡪 1 synchronization

1 🡪 n synchronization

n 🡪 1 synchronization

n 🡪 n synchronization

### Synchronization Types

There are four synchronization types:

1 🡪 1 synchronization

1 🡪 n synchronization

n 🡪 1 synchronization

n 🡪 n synchronization

Every synchronization type follows a slightly different procedure, to make sure that on assignment of one relation counterpart, the other relation counterpart goes along with it.

### 1 🡪 1 Synchronization

1 🡪 1 synchronization is quite easy. In a Jar 🡪 Lid relation, when assigning Lid . Jar, The old Jar . Lid is set to Nothing, while the new Jar . Lid is set to This.

### Risk of infinite loop 1 🡪 1

When a relation is synchronized, you may have a risk to an infinite loop.

When you assign Jar to Lid, then Lid is assigned to Jar, upon which Jar is assigned to Lid again, and so on. Fortunately, when a Jar is assigned a Lid it already has, the whole assignment is not executed. So it only goes as far as: Jar is assigned to Lid, upon which Lid is assigned to Jar again, upon which Jar is assigned to Lid again, but Jar already had that Lid, so that assignment is never executed.

### 1 🡪 n Synchronization

In a 1 🡪 n relation between Parents and Children, every time you assign a Child to a Parent, Child . Parent is overwritten. The old Child is assigned Nothing as the Parent, and the new Child is assigned its new Parent.

Also, the original Child’s ID In Parent is yielded over to the new Child. The original Child’s ID In Parent is reset.

There used to be a misunderstanding, that one Parent could reference the same Child multiple times. But that idea was abolished, because when a Parent relates to a Child twice, the Child has to relate back to the Parent twice. A Child can only have one Parent, so it can never relate back to the same Parent twice. Something like that would require an n 🡪 n relation, for the Child to be able to hold multiple references to the same Parent.

### Risk of infinite loop 1 🡪 n

Infinite loops for n 🡪 1 synchronization the are prevented the same way as for 1 🡪 1 synchronization. When you assign a Parent to a Child, The Child is added to the Parent, upon which the Parent is again assigned to the Child. But the Child already had that Parent, so the assignment is never executed. So that prevents an infinite loop there.

### n 🡪 1 Synchronization

In a Child n 🡪 1 Parent relation, when you change Child . Parent, the Child is removed from its original Parent and added to the new Parent. So you can never have the same Child in several Parents.

A Child can not appear multiple times in the same Parent, because that, in turn, should give a Child multiple references back to the Parent, but a Child holds only one reference to a Parent.

### Risk of infinite loop n 🡪 1

When a 1 🡪 n relation is synchronized, you may have a risk to an infinite loop. When you add a Child to a Parent, then the Parent is assigned to the Child, upon which the Child is added to the Parent *again*.

An earlier solution proposed for this, is that in synchronizing the relation, you never boldly Add the Child to the Parent, but you execute a Find Or Add, which prevents the Child from being added again, when it is already in the Parent’s list. This would have worked, but n 🡪 n synchronization already required a different solution, that will be more efficient for n 🡪 1 synchronization as well.

When you assign an item to a list for synchronization purposes, no synchronization is to be executed on the other side again.

You’d have to call a special List Item Set procedure, accessible only to the related class, that simply won’t synchonize back again.

### n 🡪 n Synchronization

One *related item* in one object always creates *one related item* inside the other object.

Two items, related to eachother in an n 🡪 n relation, are always connected to eachother, by connecting two specific list positions to eachother.

In n 🡪 n synchronization, Object A’s reference to Object B will be replaced by a reference to Object C. When Object B is removed from Object A’s list, then Object A is also be removed from Object B’s list. After that, Object A is added to Object C’s list.

An item in one list is aware of its position in the other list. That makes it easy for an item in one list, to remove itself from the other list.

### Risk of inifinite loop n 🡪 n

But when you add Object A to Object C’s list of related items, then Object C will try to add itself to Object A’s list of related items, upon which Object A will add itself to the list of Object C again. An infinite loop should be prevented here.

When synchronizing the relation between two objects in an n 🡪 n relation, you will add a position to the list of the referrer, and next assign an item to this position.

When you assign an item to a list for synchronization purposes, no synchronization is to be executed on the other side again.

You’d have to call a special List Item Set procedure, accessible only to the related class, that simply won’t synchonize back again.

Another solution opted at first, was to execute a Find Or Add for synchronization, instead of executing an normal Add command. That would prevent a related item from being added and added again. But then you have the problem: maybe the same item *should* be added twice to the list, because one item can relate to another item multiple times, which also requires the other item to relate back to the first item multiple times. For each reference to an item, the item must have a reference back to the referrer.

So the new option is better: you have a special List Item Set procedure, possibly called by a special Add procedure, used solely for relation synchronization, that won’t synchronize *back* again.

### The abolished multiplicity of x

Earlier I had invented a multiplicity of x, which is plural, but then a fixed set of items, for instance three items. But x can be replaced by three separate 🡪 1 relations. X was abolished in particular, because it would cause a lot of unpredictable behavior when trying to synchronize the two relation counterparts, especially in n/x 🡪 n/x synchronization.

### Confusions about relation sychronization

There used to be two points at which there was confusion about the workings of relation synchronization.

#### Confusion 1

What can be confusing is that, when a 1 🡪 n relation is synchronized, it can never be used as an n 🡪 n relation. In the relation Parent 1 🡪 n Child, every time you add a Child to a Parent, Child . Parent is overwritten. When you change Child  .  Parent, the Child is removed from its original Parent and added to the new Parent. So you can never have the same Child in several Parents. If you want to use multiple Parents, you can’t.

It is often easier to define something in 1 🡪 n relations, without thinking about it, that the backward relation might be 🡪 n too. But when a relation is synchronized, the system falls apart when a 1 🡪 n is actually intended as n 🡪 n and you’re trying to *use* it that way.

This can be misconceived as an error in the new computer language, or an inability of it, while it’s really just a wrongly defined relation.

So then it becomes really important to define the exact relation type of something, even when it’s more difficult.

If you do want to use the 1 🡪 n relation as an n 🡪 n relation, you should change the relation type, but you could also choose to split the dual relation into two unsynchronized unary relations. However, you will be loosing out on functionality and loosing integrity and coherence of the system.

A system in which all relations are dual and given the correct relation type, functions in perfect harmony and everything is logical, correct and solid.

#### Confusion 2

Synchronization could cause confusion in older versions of the computer language, where you have two dual relations to the same class, that are given the same Item Object Name. In that case, one relation’s counterpart synchronizing back to the related object could affect the other relation. By default it is not allowed to have one class A being 1 🡪 related to multiple classes, in which A has the same Item Object Name. That is only allowed if the other relations are made unary, and not dual, or if *melding* is enabled. *Melding* is a topic, which makes it possible for multiple relation classes or for instance *progressed objects (*article *Progression)*, to be referenced as a single related item, but it has to be stated explicitly that this is the intention. See the article *Melding*.

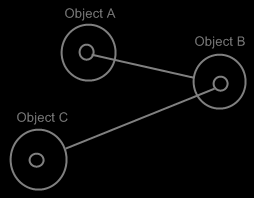
**Diagram Notation**

There isn’t really a display of relation synchronization in a diagram. You just know, that when a dual relation is expressed, the relation will be synchronized.

The procedures of relation synchronization are part of the system interface of symbols. System interfaces are the inner workings of symbols. The procedures of relation synchronization can be expressed in the diagram, if the system interfaces of the symbols are displayed. See the article *System Interface*.

This article shows diagrams to demonstrates the difference between *not* synchronizing two unary relations and *synchronizing* two unary relations.

Two unary relations will not be synchronized to eachother. An object of Class A can refer to an arbitrary object of Class B, which refers to an arbitrary object of Class A again.

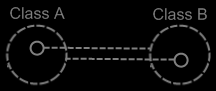


It doesn’t make the two objects refer to eachother. It just makes the two objects refer to an arbitrary object of the other class, but not necessarily to eachother.

For this, relation synchronization is applied, so that the first object and the second object always refer to eachother.



When the two following unary relations are synchronized,



then the two class lines merge together, to form the picture below:



Relations between individual objects also turn from this:



Into this:

