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

## Relations

### Related Classes

#### Concept

Target classes specify sub-objects.

If\* a sub-object in a class also gets a class assigned to it, then\* this relates the\* target class to the\* target class of the\* sub-object. The\* target classes of the\* sub-objects are called the\* object’s *related classes*.

If\* a class does not\* fix the\* class of a related item, then\* any type of object could\* be assigned as the\* related item. If\* a class fixes the\* class of a related item, then\* the\* related item can\* only become an object of that class.

A class can\* also specify related *lists*. If\* a class is assigned to this related list, the\* related list can\* only contain items of this class. If\* no class is assigned to a related list, then\* the\* related list can\* contain objects of any class.

A related list can\* also be assigned *multiple* classes, meaning that items of a fixed set of classes can\* be put inside the\* related list.

In that case one related list creates two related classes.

If\* a class’s related item does not\* have a class, the\* related item does not\* introduce a new related *class*.

#### Diagram Notation

The\* concept of related classes is explained in the\* article *Related Classes*. This article only explains its expression in a diagram.

Below are displayed an object and its class. The\* class has two sub objects, each of which points to another class:

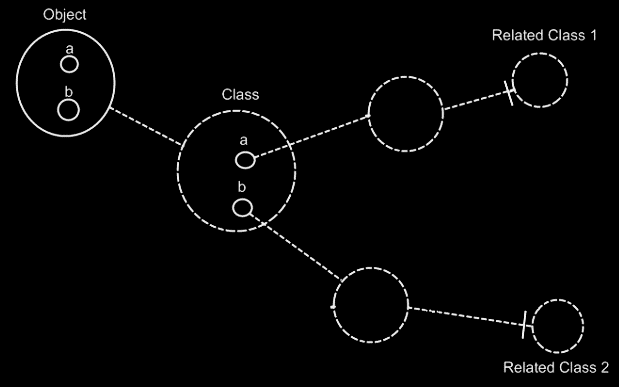


The\* **Object** automatically gets the\* same contents as the\* **Class**. The\* object’s sub-objects **a** and **b** could\* have been tied to **Related Class 1** and **Related Class 2**, but\* they are not\*, because\* the\* sub-objects of **Object** and the\* sub-objects of **Class** are implicitly tied together by the\* tie between their parents **Object** and **Class** and the\* fact, that they have the\* same name, following the\* principle of *implicitly connected through parent* (will be explained in the\* article *Automatic Containment*).

For a big part it is true, that dashes uncover the\* structure of a system, while the\* solid lines uncover the\* system’s data. All the\* classes and their related classes and the\* lines between them are dashed. However, the\* sub-objects defined inside a class are not\* dashed, because\* they do not\* function as classes themselves. And also, the\* lines from object to their class are dashed. So it is not\* 100% true, that all the\* structure elements of the\* system are dashed, and all the\* data is drawn with a solid line, but\* it’s close to it.

When\* you\* want to see the\* structure of the\* system, and ignore the\* data of the\* system, you\* just have to look at the\* diagram from the\* following perspective: class structure = classes tied together.

The\* below is the\* same example, but\* now the\* classes get further redirected.



If\* the\* dashed lines do not\* emphasize the\* classes and relations enough, a coloring could\* be applied to the\* diagram, highlighting all the\* classes and their relations to other classes.

A class can\* also have a related list: a class holds a list of items of another class. A multiplicity of *many* is expressed in the\* diagram with a nonagon:



If\* the\* nonagon is placed inside a class, then\* the\* class specifies a list of items:



No class is assigned to the\* related list here, so the\* related list can\* contain objects of any class. If\* a class is assigned to this related list, the\* related list can\* only contain items of this class.



A related list can\* also be assigned *multiple* classes, meaning that items of a fixed set of classes can\* be put inside the\* related list.



In that case one related list defines two related classes.

If\* a class’s related item does not\* have a class, the\* class has a related item, that can\* be od any arbitrary class. So this related *item* does not\* introduce a new related *class*.



### Relations

#### Concept

One object c an\* relate to other objects, but\* that’s not\* what we are usually speaking of when\* we talk about relations. When\* we talk about relations, we are talking about relations between *classes*. Relations between classes set the\* configuration of how objects can\* be connected to eachother. Classes and their relations determine the\* rules by which the\* objects behave.

The\* concept of *relations* is about thinking in relations between classes, rather than loosely tying together arbitrary objects.

A class functions as a blueprint for objects. The\* class structure determines which types of objects can\* be connected to each other, but\* not\* yet what specific objects are connected to eachother. Which specific objects are connected to eachother is determined by the\* object structure. The\* class structure only defines which types of objects can\* be connected to eachother.

The\* class-relation structure is the\* bonestructure of a program.

The\* article *Related Classes* already explained how one class relates to other classes. But\* this is a very single-sided view on relations. It does not\* create a relation from the\* related class back to the\* first class again.

Relations are usually bi-directional. If\* one class relates to another, then\* the\* other class relates back to the\* first class. So next to one class getting a sub-object of another class, the\* other class also has to get a sub-object that points back to the\* first class.

You\* can\* also have a **1 🡪 n** relation between two classes. In that case one class has a related list of items of another class. The\* other class has a single related item, that connects back to the\* first class. There are also **n 🡪 n** relations, where one class holds a related list of items of another class, and the\* other class also holds a list of related items, that connects back to the\* first class.

#### Diagram Notation

It must be mentioned, that the\* method of *automatic containment* applies to unary references, and just *does not\* match* yet with the\* notation for relations. In the\* future it must be further worked out how to best turn them into a single notation.

As explained in the\* article *Related Classes in a Diagram*, a relation

between one class and another can\* look like this:



Because\* Class A has a sub-object of **Class B**, this creates a relation from **Class A** to **Class B**. However, **Class B** does not\* have a relation back to **Class A** yet. The\* picture below, adds the\* relation back to **Class A**:



Because\* the\* class references back and forth are so closely related, the\* two class lines merge together to form the\* picture below:



This, however, creates an ambiguity in the\* notation. The\* two circles tied together with a class line suggest, that they are both the\* same class. But\* the\* circle inside **Class A** represents **Class B** and the\* circle inside **Class B** represents **Class A**.

Fortunately, the\* notation can\* be disambiguated using the\* rules of automatic containment. Automatic containment is explained in the\* article *Automatic Containment*. Before explaining how automatic containment leads to the\* eventual notation, here is the\* disambiguated notation of a relation between two classes:



The\* notation is accomplished by first taking the\* original picture with one class refering to another and the\* other refering back to the\* first class:



Then\*, an imaginary reference to each class is added to the\* diagram



Next, the\* class lines are merged, but\* also the\* class symbols are merged:



The\* notation would\* still be ambiguous, if\* it weren’t for the\* double dashed line of the\* merged class symbols. So a double dashed circle symbolizes a relation between classes.

The\* picture above expresses a 1 🡪 1 relation between **Class A** and **Class B**. But\* other multiplicities can\* also be used. The\* multiplicity of **n** is expressed with a nonagon:



A nonagon represents a list of things. Instead of letting a **Class A** contain a single item of **Class B**, you\* can\* let is contain a list of items of **Class B**:



The\* picture above expresses an **n 🡪 1** relation between **Class A** and **Class B**.

The\* picture below displays a **1 🡪 n** relation between **Class A** and **Class B**.



Finally, the\* picture below displays an **n 🡪 n** relation between **Class A** and **Class B**.



A symbol merge in a relation that has nonagons in it also results in a double dashed circle, because\* the\* imaginary reference to the\* classes, that are put on a higher level, are represented by circles, not\* a nonagons.

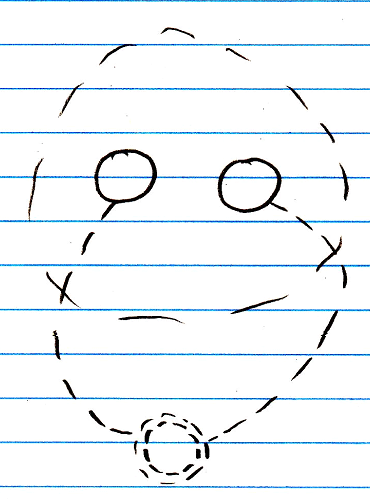
One related list can\* also contain items from multiple classes. **Class A** has a **n 🡪 1** relation to items of **Class B** and **Class C**, it is expressed as follows:



#### Class relating to itself

A class can\* have a relation to itself. For instance, a person can\* relate to a parent, which is also a person. So a person is related to a person, which relates a class to itself.

A class relating to itself looks as follows in a diagram:



#### Counterpart out of sight

When\* the\* counterpart of the\* relation is out of sight, a line should point out of the\* diagram. A catch there is, that you\* can\*’t see if\* the\* relation counterpart is part of a multiplicity of **n** or not\*. Therefore, the\* multiplicity is expressed at the\* end of th line pointing out of the\* diagram as follows:





#### No reuse of merged imaginary references

If\* two imaginary references have merged, to become a relation symbol, then\* other references to the\* same classes won’t connect to an imaginary reference that has merged to become a relation symbol. Relations create their own imaginary references, that aren’t reused. This is displayed in the\* article *Relations Between Objects in a Diagram*, but\* may also apply to the\* notation of relations between classes.



#### Example

Classes and their relations define the\* behavior of your system, so it is very important to be aware of them, instead of just looking at individual objects, tied to other objects.

The\* example below is part of the\* class-relation structure of a drawing program.

It displays the\* classes **Application**, **Document**, **Point** and **Line**.



An instance of the\* application can\* hold multiple open documents. So **Application** has a **1 🡪 n** relation with **Document**. That automatically makes a document part of one instance of the\* application. A document holds a collection of points and a collection of lines. That makes **Document** have a **1 🡪 n** relation with **Point** and a **1 🡪 n** relation with **Line**. This automatically makes **Points** and **Lines** part of a single document. Furthermore, a line is composed of two points. However, if\* you\* chain lines together, a point can\* become a part of multiple lines. This gives **Line** two **1 🡪 n** relations with a **Point**: one for the\* first point and one for the\* second point.

The\* example displays all the\* classes, relations, related items and related lists of the\* object structure.

If\* just the\* use of dashed lines does not\* emphasize the\* classes and relations enough, a coloring could\* be applied to the\* diagram, highlighting all the\* classes and relations.

### Dual & Unary

#### Concept

A unary relation is a relation with only one direction: one class relates to another class, but\* the\* other class doesn’t relate back to the\* first class.

A dual relation is a bi-directional relation: one class relates to another class, and the\* other class relates back to the\* first class.

In most cases, it is best to make a relation dual. You\* don’t even have to give the\* the\* backward related item a name, just let it sit there, until you\* find a name for it.

Bi-directional relations were already introduced by the\* article *Relations*. What is left to cover is the\* reason why to make a relation dual or unary.

#### Ridiculous to maintain backward relation

Only if\* storage of a relation counterpart results in a ridiculous amount of data, that you\* don't even use, then\* you\* may want to omit the\* backward relation.

This is the\* case when\* the\* target class of the\* relation is very generally used. The\* key example for this is a **Number**. A number is used by too many other classes, so it is ridiculous to give a **Number** a related list for every class that uses **Numbers**. To determine if\* a backward relation is ridiculous to maintain you\* could\* also consider the\* following:

- No functional correspondence with anything particular

- Too many objects will refer to this.

This is all very subjective, but\* I can\*’t give a more exact definition for it.

#### Unable to program class

Another reason for not\* keeping the\* backward relation, is that you\* may not\* be able to program the\* target class, because\* somebody else authored it. But\* there’s a way to go around this: use inheritance to create a derived class, relate to the\* derived class, storing the\* backward relations inside the\* derived class. Then\* the\* original class is not\* burdened with extra related lists. The\* derived class is an extension of the\* original class.

#### The\* Referrers Concept

A **Number** class could\* choose to support the\* **Referrers** concept. This will give a **Number** object only one list of all referrers, instead of a separate list for every class that uses **Numbers**. **Numbers** may be used by many classes, but\* an individual **Number** object, is never used much. It is not\* a lot of data to register inside an **Number** object, which objects refer to that particular **Number**.

But\* then\* the\* **Number** class will also register all its *class referrers*, which is undoable, because\* a humungous amount of objects refer to this class. But\* a solution for this was already proposed by the\* article *Class Referrers*. You\* can\* choose for a class to not\* register its class referrers, while objects do register their referrers.

#### No exact formula

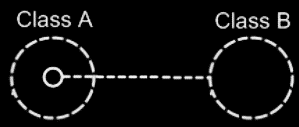
I admit, that I would\* like to give an exact formula for when\* a backward relation should or should not\* be maintained. But\* for now, I can\* only give a functional description of when\* it is ridiculous to maintain a backward relation. A programmer is going to have to determine it, when\* a relation should be unary.

#### Diagram Notation

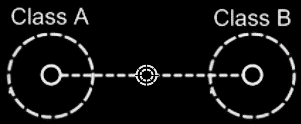
The\* concept of dual and unary is explained in the\* articles *Dual & Unary, Related Classes* and *Relations*.

The\* expression of dual and unary in a diagram has already been introduced in the\* articles *Related Classes in a Diagram* and *Relations in a Diagram*. This article only puts the\* diagram expressions of dual and unary next eachother for a comparison.

This is a unary relation between **Class A** and **Class B**:



This is a dual relation between **Class A** and **Class B**:



### Relations Between Objects

#### Concept

Relations between classes set the\* rules for how objects can\* be connected to eachother.

Relations between *objects* are the\* actual connections.

A relation between objects is always a relation between *two* individual objects. If\* one object refers to another, the\* other refers back to the\* first one.

From a 1 🡪 1 perspective this seems logical, but\* from an n 🡪 n perspective this may not\* seem logical.

A relation between two classes with each a multiplicity of **n**, creates a related list in both of the\* classes. Every object of those classes will contain a related list of related items.

Any object that as a relation to another object, gives the\* other object *one* relation back to the\* first object. So for each reference to an object, the\* other object contains a reference back. One reference inside an object is tied to one reference inside another object.

One object can\* relate to multiple objects, so an object *can\** have a one-to-many relation to other objects, but\* one *related item* in one object always creates *one related item* inside the\* other object.

#### Diagram Notation

The\* main rule about relations to object is: for each reference to an object, the\* other object contains a reference back.

When\* you\* draw out the\* separate counterparts of a relation between two objects it looks like this:



But\* to express the\* close relation between the\* two references, the\* two lines are merged, and disambiguated from normal object lines with a *relation symbol*:



The\* relation symbol is a double circle. The\* reason behind this notation, was already explained in the\* article *Relations in a Diagram*. The\* notation is accomplished by first taking the\* original picture with one class refering to another and the\* other refering back to the\* first class:



Then\*, an imaginary reference to each class is added to the\* diagram



Next, the\* class lines are merged, but\* also the\* class symbols are merged:

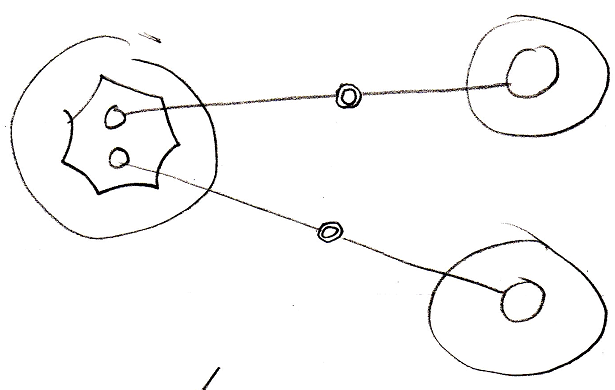


The\* notation would\* still be ambiguous, if\* it weren’t for the\* double line of the\* merged object symbols. So a double circle symbolizes a relation between objects.

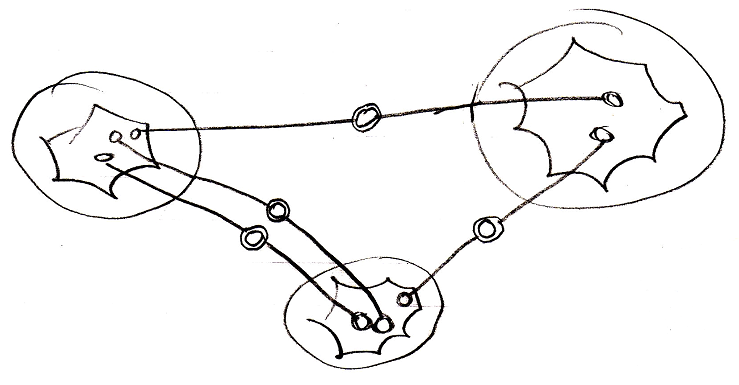
For relations between classes the\* relation symbol is a double *dashed* circle. For relations between objects, the\* relation symbol is a double circle drawn with *solid* lines.

In 1 🡪 n and n 🡪 n relations the\* rule, that each reference contains one reference back, also applies:

1 🡪 n:



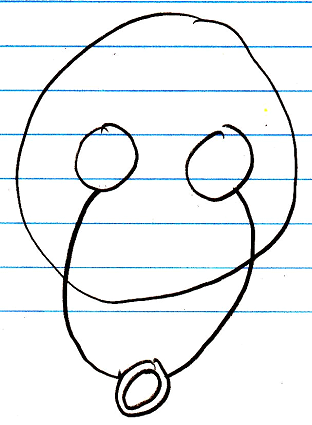
n 🡪 n:



It doesn’t matter whether an object reference is part of a list or not\*: every time it is still *two* individual object references, that are tied together.

#### Object relating to itself

Sometimes an object relates to itself. In a diagram this looks as follows:



#### Counterpart out of sight

When\* the\* counterpart of the\* relation is out of sight, a line should point out of the\* diagram. A catch there is, that you\* can\*’t see if\* the\* relation counterpart is part of a multiplicity of **n** or not\*. Therefore, the\* multiplicity is expressed at the\* end of th line pointing out of the\* diagram as follows:



#### No reuse of merged imaginary references

If\* two imaginary references have merged, to become a relation symbol, then\* other references to the\* same objects won’t connect to an imaginary reference that has merged to become a relation symbol. Relations create their own imaginary references, that aren’t reused.

Here is a relation between two objects:



The\* two objects refer to eachother. This originally consisted of two distinct references:



Imaginary references were put on one level higher:



If\* other references to the\* same objects were also displayed in the\* diagram, then\* they would\* connect to the\* same imaginary references, put on a higher level:



When\* you\* merge the\* imaginary references to display that two references are part of a single relation, you\* will not\* connect all

references to the\* merged imaginary reference:



You\* will keep separate imaginary references for the\* other unary relations to the\* objects:



### 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*.

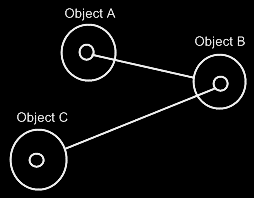
#### Diagram Notation

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:



## Loose Ideas

### Loose Ideas about the\* Relations Section

#### ± 2004

The\* main unit in a relational structure is the\* *class*. There is a list of *classes*.

The\* classes are tied together with *relations*. One class is tied to another.

The\* main object that defines a relational structure is the\* Structure object. The\* Structure object contains a Classes collection and a Relations collection. If\* you\* want to add a class, you\* do that in the\* Classes collection. If\* you\* want to add a relation, you\* do that in the\* Relations collection.

The\* Classes collection contains objects of class Class.

The\* Relations collection contains objects of class Relation.

Every Class has an Attributes collection. A Person Class, for instance, could\* have a Name Attribute and an EmailAddress Attribute and more attributes such as Street, HouseNumber, ZipCode, etcetera. A Class also contains a RelatedClasses collection, which reflects all of the\* class’s related classes. You\* can\*’t add RelatedClasses to this collection. You\* have to define relations in the\* Structure.Relations collection and they will be *reflected* in the\* Class.RelatedClasses collection. Other members of the\* Class class are explained in other sections, covering different concepts. However, all members are briefly explained in a sub section below.

A **Relation** consists of two **RelationClasses**. It contains two **RelationClass** objects that define the\* two classes of the\* relation and how they relate to one another.

A **Relation** also defines whether the\* relation is **Dual** or **Unary**, by the\* **Boolean** **Dual** member. If\* a **Relation** is **Dual**, then\* both classes are aware of eachother and refer to one another. If\* a **Relation** is **Unary** then\* only **RelationClassA** is aware of **RelationClassB** and refers to it, but\* **RelationClassB** is unaware of **RelationClassA** and doesn’t refer to it.

Even though a **Dual** relation would\* seem to make **RelationClassA** and **RelationClassB** equal opponents, **RelationClassB** in many cases is the\* inferior one. For instance, in writing XML files, **RelationClassB** is seen as contained in **RelationClassA** and not\* the\* other way around. In that case *direction* of the\* relation does matter. However, still in many cases **RelationClassA** and **RelationClassB** are technically equal opponents. When\* you\* keep in mind which **RelationClass** is inferior and which one is superior, things like XML writing go well automatically. If\* you\* are sloppy with choosing if\* something is **RelationClassA** or **RelationClassB**, you\* could\* get trouble that makes you\* obliged to switch the\* two relation classes within the\* relation, but\* usually you\* won’t notice anything going wrong. So relax, but\* beware.

The\* two **RelationClass** objects define the\* relation furtherly.

The\* member **Class** of **RelationClass** is very important to set, and it defines which class makes part of the\* relation. Define the\* **Class** in both **RelationClasses** of the\* **Relation** and you\*’ve made a relation between the\* two classes.

A very important member of a **RelationClass** is the\* **AbstractNumber**. This defines whether a relation class is **1**, **x** or **n**. If\* you\* define the\* **AbstractNumber** for both of the\* two relation classes, you\* can\* for instance make a 1🡨🡪n relation between the\* two classes or a x🡨🡪n relation or whatever. **AbstractNumber** is **1** by default.

To define the\* quantity of x, you\* set **ExactNumber**. For instance, in a   
**Line n🡨🡪2 Point** relation, you\* define for the\* **Point** **RelationClass** that its **AbstractNumber = x** and its **ExactNumber = 2**.

Two other important members of a **RelationClass** are **CreateObjects** and **EnsureSubObjects**. **CreateObjects** is by default **True**, exceptions not\* regarded. **CreateObjects** says that when\* a new position is created within the\* related list, an object is instantly *created* in that position (See *Objects and Object Positions*). In many cases you\* want that to happen. Sometimes you\* don’t want objects to be created, because\* you\*’d want to assign an object to that position yourself. Then\* you\* set **CreateObjects** to **False**. Furtherly, **EnsureObjects** will see to it that you\* can\*’t assign **Nothing** to the\* object position. It is **True** by default, but\* can\* be set to **False**. For more information see the\* sections *Create Objects* and *Ensure Objects*.

And then\* there’s another member of **RelationClass** that is important to mention. And that is **ListType**. **ListType** is usually set to **NormalListType**, but\* can\* also be set to **ReferenceCountedListType**, **RegistrationListType**, **SharedListType** or **SelectionListType**. The\* list then\* gets very special behavior. For more information see the\* *Specialized Lists* section. Although I don’t explain them here very thoroughly, **ListType** does mean a lot for the\* general structure of the\* system.

You\* can\* see that inside a **RelationClass** object, much more is defined than just the\* **Class**. That’s why **RelationClass** is a separate class. Many times I will speak of a relation class, and you\* shouldn’t confuse it with just a class then\*, because\* it will be defining a class as it is in the\* context of a relation.

The\* elements as I’ve described them in this section, form the\* following structure of Classes, Attributes and Relations.

Structure

|

|-- Classes

| |

| |-- Class ()

| |

| |-- Attributes

| | |

| | |-- Attribute ()

| |

| |-- RelatedClasses

| |

| |-- RelationClass ()

|

|-- Relations

|

|-- Relation ()

|

|-- Dual

|

|-- RelationClassA and RelationClassB

|

|-- Class

|-- AbstractNumber

|-- ExactNumber

|-- CreateObjects

|-- EnsureObjects

|-- ListType

Every other concept of J Data is hung up on this main structure of **Classes**, **Attributes** and **Relations**.

For instace, the\* physical appearance in the\* user interface of a list defined in a **RelationClass** with **AbstractNumber = n**, is defined inside that **RelationClass** object. The\* whole appearance of a **List Control** is defined inside a **RelationClass**. That is an example of how the\* user interface is defined right inside the\* general structure above. The\* same way *all* of the\* application’s features are defined in the\* context of the\* relational structure of **Classes**, **Attributes** and **Relations**.

### Loose Ideas about Dual & Unary

Relations,

New thing: what I should consider in the\* future, is that a relation counterpart can\* be completely derived from the\* other relation counterpart. Therefore, you\* might make a relation dual, so the\* counterpart usable, but\* not\* STORE it, but\* derived it somehow. That way you\* can\* use all relation counterparts, just not\* store the\* ridiculously large ones. > No, because\* then\* you\*’d have to scan the\* whole internet for referrers.

JJ

### Loose Ideas about Relations in General

*The\* texts below are loose ideas yet to be turned into good documentation.*

References,

2008-11-05

It is important to exactly see

which objects can\* be accessed through an object.

You\* should see access connectors for them.

It is also very important that you\* can\* exactly see

which object access something.

You\* have too much the\* ability to not\* register

which objects actually access something.

Perhaps in practice it is not\* so bad to impose registering

dependencies always. Perhaps practically the\* consequences

are overviewable.

The\* negatives about not\* seeing ALL referrers, but\* only some,

or optionally are BAD. Because\* not\* seeing the\* connections between

all things create a lot of problems in software systems today.

Perhaps most problems with software systems today have to do

with not\* knowing what exactly makes use of what.

But\* how about commonly used classes, such as integer.

Integer objects all around can\* store a link to the\* integer class

on the\* computer language site. But\* the\* integer class

on the\* computer language site can\* not\* register all objects

around the\* globe of class integer.

Or perhaps consequences of many many references to the\* same

class can\* be MADE overviewable.

Perhaps you\* can\* make intermediate references to class integer

on your local site or local module. Then\* the\* references to class integer

on that site, reference the\* local reference to class integer.

The\* references to class integer on a site are registered in

the\* local site's shadow of class integer.

And in class integer on the\* computer language site only the\* shadow

itself is registeren.

Perhaps you\* can\* enforce such a pattern.

It is always a problem with classes widely used.

Any class could\* potentially be widely used.

You\* could\* set reference quota, though, to protect your site.

But\* class integer should be used billions and billions of times.

Perhaps to protect your site, you\* make a reference quata,

or you\* enforce shadowing.

In a shadow situation, I'd like to also see how many referrers

a shadow of class integer has. But\* you\* can\* do that.

Site computer language has class integer, which registers all

sites using class integer, and those registrations consist of

the\* registration of a shadow reference of class integer,

and the\* shadow reference of class integer returns the\* references

of the\* shadow again, but\* those references are stored on the\*

client site, not\* on the\* computer language site.

You\* still register all integers, but\* the\* registration is spread over

multiple sites, so the\* costs are spread and everybody pays

a reasonable amount of storage cost.

But\* could\* this pattern be misused?

What if\* a new internet protocol allows many many more sites,

and somebody thinks it is cool to create 1,000,000 virtual sites,

for some purpose and each site shadows class integer.

Then\* you\* have 1,000,000 more registrations in class integer.

That's where quota's come in.

But\* that can\* also be abused. An attack could\* use up the\* quota,

and new shadows to integer can\* not\* be made anymore.

Existing sites, that use class integer still work,

but\* nobody can\* program a new site of class integer then\*.

But\* there is a difference between in good practice, and in bad practice.

You\* have to ask yourself: how can\* we make it practically work when\* we

are all behaving ourselves. Some things do not\* work practically even if\*

we do behave ourselves. That's one area of problems to work on.

It is another area of problems where well behaved practice works,

but\* bad behavior overthrows the\* system.

That last part we call attacks, virusses, threats, etcetera.

That area of problems should be adressed differently.

It is a principle, that good practice should be facilitated,

and bad behavior should not\* compromise how clear you\* can\*

organize your system, so should not\* compromise your freedom.

Bad behavior should be adressed separately in the\* background.

Enough for now.

JJ

Taken out of Interfaces Articles on 2010-05-07:

Preventing class’s extension with commands:

A class can\* prevent itself from getting further extended with commands.

For instance, you\* don’t want the\* class Integer to be extended with any command that uses an integer. It is a question of it being rediculous to maintain a list of all commands that uses integers. Don’t prevent a class from being extended with commands, just because\* you\* think it makes your interfaces more reliable. Class Integer can\* also just make it a *default*, that it doesn’t get further extended with commands. Some commands that use Integer, you\* might indeed want to see added to class integer, but\* you\* need to actively choose that then\*. If\* parameters don’t extend a class with a commands, these parameters are only shown as parameters, and not\* as commands inside a class definition.

> 2008-09-06 This is actually a non-dual relation.

This is basically the\* same issue as covered in the\* section Dual & Unary, which explains, when\* it is ridiculous to maintain a backward relationship. For instance: relationships from class Integer back to any class, that uses an integer are rediculous to maintain as well.

And if\* changes to the\* class are ventilated to all the\* objects this happens with the\* help of events, but\* usually you\*’ll just use versioning to use an unchanging production version of a class.

a class can\* *access control* the\* fact whether dual relations can\* be established to it.

I think, that a dual relation has a source site. One end of the\* relation has authored the\* relation, the\* other relation just complied.

Actually, this could\* happen dually.

JJ

Backwards relation alternative: Site usage

2010-05-03

For ‘ridiculous to maintain’ you\* could\* also go with this approach: make the\* small thing, like Integer always referenced in a qualified way through the\* site and maintain a unique list of source sites. That would\* at least give you\* an idea of how many sites are used and if\* any sites still use it.

JJ

Relationships,

2008-11

The\* roles that symbols get when\* connecting them with lines.

JJ