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

## Relations

### Concept

One object can 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 eachother, 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.

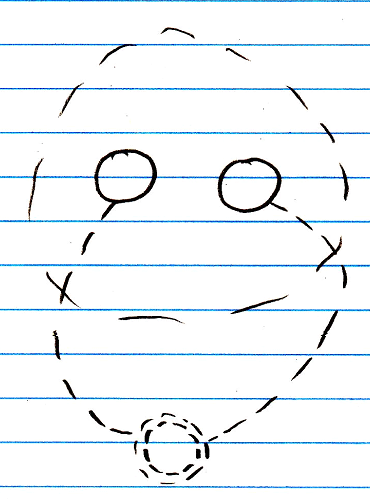
One related list can also contain items from multiple classes. Class A has a n 🡪 1 relation to items of Class B and Class C, it is expressed as follows:



### 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.

## Ideas

### ± 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**.