|  |
| --- |
| Circle Broader View |

## Relations

### Relation Direction

When\* all relations are bidirectional, a side-effect might be that everything might end up at the\* same level hierarchically, since all the\* relationships are mutual.

That might bump with the\* containment structures that might look nice in Circle notation.

A proposed solution might be to specify a direction to these relations, so that the\* 'inferior' part of the\* relationship might be put at a lower level of containment, restoring the\* use of a containment structure. Perhaps a 1 to n relations might already imply direction: parent on top, children below.

This may only be a problem, if\* the\* containment structure would\* have to be figured out by the\* system on its own.

Another solution might be that a programmer can\* pick the\* containment level, so then\* maybe it is not\* really a problem and relation direction would\* be implied by the\* containment levels picked by a programmer.

### Ubiquitous Bidirectional Relationships

#### Concept

In most cases, it is best to make a relationship bidirectional. 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.

Bidirectional relationships were already introduced by the\* article *Relationships*. What is left to cover is the\* reason why to make a relationship bidirectional or unidirectional.

#### Ridiculous to maintain backward relationship

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

This is the\* case when\* the\* target class of the\* relationship 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 relationship 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 relationship, 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 relationships 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.

#### No exact formula

I admit, that I would\* like to give an exact formula for when\* a backward relationship 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 relationship. A programmer is going to have to determine it, when\* a relationship should be unidirectional.