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

## Relationships Broader View

### Relationship Direction

When\* all relationships 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 relationships, 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 relationships 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 relationship 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\* 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.

## Loose Ideas

### Loose Ideas about Ubiquitous Bidirectional Relationships

Relationships,

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

JJ

-----

References,

2008-11-05

It might be relevant to see which objects could be accessed through an object.

Maybe show access connectors for them.

It might also be relevant that to 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 registered.

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 quota, 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

-----

Backwards relationship 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

-----

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-bidirectional relationship.

This is basically the\* same issue as covered in the\* section Bidirectional & Unidirectional, 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 ridiculous 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 bidirectional relationships can\* be established to it.

I think, that a bidirectional relationship has a source site. One end of the\* relationship has authored the\* relationship, the\* other relationship just complied. Actually, this could\* happen dually.

JJ