|  |
| --- |
| Circle Language Spec: Interfaces |

## Interfaces Edge Cases

### Member Grouping

Triangles could be used to indicate a benign member grouping:

|  |  |  |
| --- | --- | --- |
|  | 🡪 |  |

Because a triangle basically merges with its container, it is like you are not bothered by the borders of a triangle.

At first glance the grouping does not seem to affect how the objects are used. However, two problems arise:

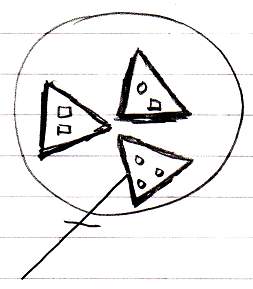
- Explicit interface dependency

- Call overhead

Below these two problems are explained and possible solutions are given. Solutions for these problems are still very much open to discussion.

*Explicit Interface Dependency*

*Existing* functionality may not be affected by member groupings. However, you could make *new* connections directly to the new interface objects.



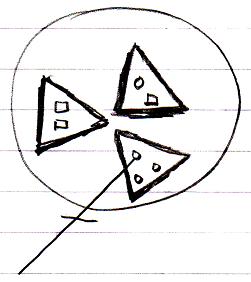
The new connections are now dependent on the member grouping, which was only meant to be a loose definition.

To allow member grouping to be benign, some solution must be found for this. The solution probably lies in the area of access controlling system commands. The member groupings need to be access controlled enough to not be able to explicitly connect to them, but not so much that it makes their members totally inaccessible. Separately access controlling Get For Access and Get For Copy might solve this problem. However, it needs to be thought through if this really only blocks something’s usage as a qualifier. If it does not, probably another split up in system commands will do the trick.

Ideas about the separation between Get For Access and Get For Copy can also be found in the *Black Box* and *System Objects* chapters.

*Call Overhead*

Member grouping may seem benign, but when you approach a member of a group, an Object Get command on the group must be executed before accessing the member, which causes (a small) processing overhead on the call.



The solution could either lie in *optimization*, or making the triangles *imaginary*.

Imaginary elements of a diagram were introduced in the *Automatic Containment* article. Making the member groupings imaginary might be a bad idea, because up until now imaginary elements of the diagram were automatically calculated by the system, and not set by a programmer and it might be wise to keep it that way.

A better solution may lie in optimization. Programmers often add structure to their programs even though it is not absolutely necessary, but just makes things more tangible and orderly. Taking such structurings out of a program before it is run, is more of an optimization issue than anything else.

### Interface Referrers

The *Referrers* concept has been explained in the *Relations* chapter. The article *Class Referrers* explains the concept for classes. It works exactly the same for interfaces. It will not be further explained here. Mind that commands can have interface referrers as well.

### Reliability of Interfaces

An interface is a contract. A welcome fact about a contract is that once agreed upon, it does not change. So an interface should be reliable.

However, the concept of *relations* can make interfaces unreliable.

Relations are bidirectional. When something points to something else, the other thing points back. This causes extra members to be dynamically added to the interface of another object. The interface of one object changes when other objects start pointing to it.

The benefits from bidirectional relations are too important to rid of. They solve a difficult fundamental problem in computer technology. A lot of times you want to know whether an object is still used and what specifically still uses this object. In computer systems today this is often difficult to find out. Knowing all backward connections all the time would solve this problem.

Bidirectional relations also see to it that when a command has a parameter, the command is immediately available from any object that the command could be executed on.

To keep interfaces reliable, somehow the backward relations should not become part of the interface. The solution is to make the backward relations Private. This excludes the members from the interface, but keeps them available as part of the implementation.

Other options have been proposed to responsibly get rid of the automatically created members. The options will be explained now.

Backward relation options:

- Private

- Public

- Manual

- Conditional

- Preliminary

#### Private Backward Relation

Allowing backward relations but making them Private means the backward relations are maintained, but they do not become part of the interface.

#### Public Backward Relation

Allowing Public backward relations means the interface will not be stable, because extra members are created when things start pointing to an object. However, this does not have to be a problem in all cases.

#### Manual Backward Relations

In this case backward relations are not created automatically at all. Logically this keeps the interface stable. You can program the backward relation later, but only if you know where the foreword relation is. To find all foreword relations you may need to scan the entire earth, so it is not easy.

When you are able to program the relation back, you also need permission from the foreword relation, because this could have great implications on the way the other object functions.

Maintaining a backward relation should be strongly stimulated because it solves such a difficult fundamental problem in computer technology. To equally honor the values of both the interface concept and the relations concept, the default solution should be to make backward relations Private. Keeping backward relations Private keeps interfaces stable.

Now follow some other solutions, that are sort of arbitrary and miscellaneous, but possibly handy in practice.

#### Conditional Backward Relations

Perhaps you should automatically accept backward relations based on a condition, for instance that the relation counterpart is within the same module. This may be a neat editing feature. That way you can program a module and let backward relations build up, but outside sources can not influence this process. This limits control to interface extensions to the module itself. In a broader sense you are defining a condition for automatically accepting a backward relation: namely that the relation counterpart must be inside the same module.

#### Preliminary Backward Relation

Preliminary backward relation means that the forward end of the relation will maintain the backward relation until the relation counterpart either accepts or refuses it. This politely keeps the other related object’s interface and storage in tact while still offering the ability to adopt the index. You may also solve a quota overrun security problem this way.

This sounds good in theory, but in practice it may turn out that the registration of the request takes the same amount of storage as storing the backwards relation itself.