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

## Relationships

### Introduction

This topic aims to introduce that the build up of classes might be viewed as a model of relationships between classes. Also, bidirectional relationships might be introduced. A notation for those is suggested and a description of what that might mean. An attempt is also made to describe how this compares to relations between individual objects.

### Relationships Between Classes

One object may relate to another object, but that might not be what the topic of relationships commonly is about. It might be more about relationships between *classes* than individual objects. Those may determine the configuration of how objects are connected to each other, rather than just loosely tying together arbitrary objects.

#### Concept

A class might specify a configuration of sub-objects. This is an attempt to summarize the kinds of relationships between classes this might create.

A sub-object inside a class might have a class. This would relate these two classes together.

When a class would set the class for a related item, the related item might only ever be an object of that specific class. When a class would not set the a related item's class, then any type of object might be assigned to it. That would *not* introduce a relationship between classes.

A class might also specify related *lists*. When a class would be assigned to this related list, the related list might only contain items of this class. When no class was assigned to the related list, then the related list might contain objects of any class.

There is also the idea for a related list to be assigned *multiple* classes, which might mean that items of a fixed set of classes may be put inside this related list. In that case one related list may create two relationships between classes.

#### Diagram Notation

Below an attempt to depict an object and its class.



The **Class** would contain two sub-objects, each pointing to another class. The **Object** would get contents similar to the **Class**. To 'see' the relationships between classes, it might be an idea to focus on the dashed shapes and then the dashed lines in between them. (In case of different choices in using dashed shapes, it may be more difficult to see the relationships between classes in a diagram.)

The class of a related item might not be set.



Then **Related Item** might have any class. It might be interpreted as: this related *item* does not introduce a related *class*.

A class might also have a related list: a class would then holds a list of items of another class. A multiplicity of *many* may be expressed in a diagram with a nonagon symbol:



A nonagon might be placed inside a class, which may imply that the class specifies a *list* of items:



When no class is assigned to the related list, it might imply that the related list could contain objects of any class. When a class *is* assigned to to the related list, it suggests the related list might only contain items of this class.



There is also an idea that a related list might be assigned *multiple* classes, which might mean that items of a fixed set of classes could be put in the list.



In that case one related list might define two related classes.

### Bidirectional Relationships

#### Concept

The relationships described so far would be *unidirectional*: one way only. Relationships might also be *bidirectional:* a two way street. Then if one class relates to another, then the other class might relates back to the first class again. In case of a bidirectional relationship if one class gets a sub-object of another class, the other class would also get a sub-object pointing back to the first class.

There might also be **1 => n** relationships between classes. In that case one class may have a related list of items of another class. The other class might have a single related item, that connects back to the first class. There may also be **n => n** relationships, where one class would hold a related list of items of another class, and the other class might also hold a list of related items, that connects back to the first class.

#### Diagram Notation

A unidirectional relationship between one class and another might look like this:

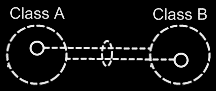


**Class A** has a sub-object of **Class B**. This would create a relationship from **Class A** to **Class B**.

**Class B** does not seem to have a relationship back to **Class A** yet. The picture below would add that relationship back to **Class A**:



Because the class references back and forth seem so closely related an additional to the notation is proposed here. The two class lines may be joined together with a relational ring:

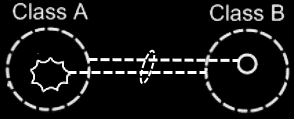


(The the relational ring's being dashed might be a stylistic choice. A solid line type or a slightly tilted ring shape might be variations.)

The picture above would express a 1 => 1 relationship between **Class A** and **Class B**. But other multiplicities might also be used. A multiplicity of **n** might be expressed with a nonagon:

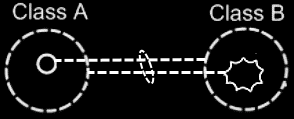


A nonagon would represent a list of things. Instead of letting a **Class A** contain a single item of **Class B**, it may contain a list of items of **Class B**:

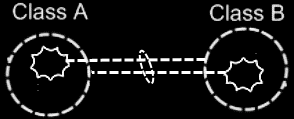


The picture above aims to express an **n => 1** relationship between **Class A** and **Class B**.

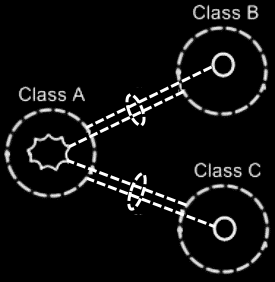
The picture below would display a **1 => n** relationship between **Class A** and **Class B**.



The picture below aims to display an **n => n** relationship between **Class A** and **Class B**.



There is an idea where one related list might contain items from multiple classes. **Class A** would have a **n => 1** relationship to items of **Class B** and **Class C**, which the picture below aims to express:



#### Class Relating to Itself

A class may relate to itself. For instance, a person might relate to a parent, which could also be a person. So then a person would be related to a person, which might relates a class to itself.

A class relating to itself could look as follows in a diagram:

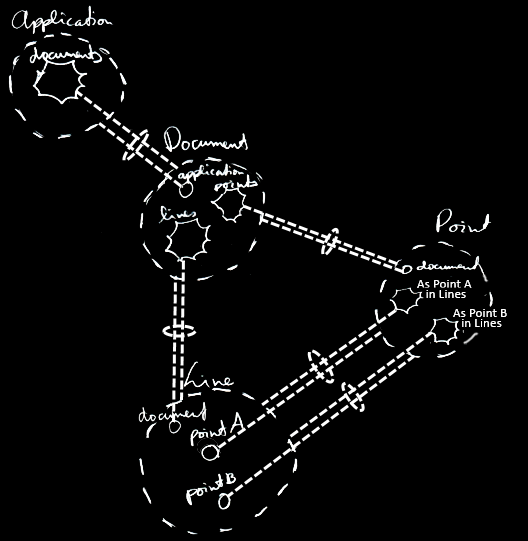


#### Example

Classes and their relationships might define behavior of a system, so it might be relevant to be aware of them, instead of looking at individual objects, tied to other objects.

The example below could be a piece of the class-relationship structure of a drawing program.

It aims to display the classes **Application**, **Document**, **Point** and **Line**.



Here is an attempt to describe with text the classes and relationships that would be drawn out in the picture: A running **Application** could hold multiple open **Documents**. So **Application c**ould have a **1 => n** relationship with the **Document class**. Viewed in the opposite direction, this would make a **Document** opened inside a running **Application**: an **n => 1** relationship between **Document** and **Application**. A **Document** might hold a collection of **Points** and a collection of **Lines**. That would make **Document** have a **1 => n** relationship with **Point** and a **1 => n** relationship with **Line**. Inversely, this would make collections of **Points** and **Lines** part of one **Document**. Furthermore, a **Line** would be composed of two **Points**. However, when chaining lines together to form a picture, a point might be reused as part of multiple lines. This should give the **Line class** two **1 => n** relationships with a **Point**: one for the first point and one for the second point: **Point A** and **Point B**. The *inverse* relationship, could be that each point may be used in multiple lines, as either **Point A** or **Point B**. Perhaps a bit creatively, these became collections called **As Point A In Lines** and **As Point B In Lines**. It may be a bit non-obvious that it would be a **1 => n** relationship, involving lists of lines.

The example aimed to display all the classes, relationships, related items and related lists of the class structure.

### Bidirectional Relationships Between Objects

#### Concept

Relationships between *classes* may set guidelines for how objects might connect to eachother.

Relationships between *objects* would bethe *actual* connections between objects.

Something special might happen in the translation of relations between classes to relations between objects. But the mechanics might also overlap.

The bidirectional relationships between objects all seem be **1** to **1** in a away: relationships between pairs of objects. If one object would refer to another, the other one might refer back to the first one.

What happens with **1** to **n** and **n** to **n** relationships between classes in this case, is that those might result in multiple **1**to**1** relationships between objects.

Relationships between classes with a multiplicity of **n** might create related lists inside classes. In objects of those classes these lists might contain separate objects. Any object that as a relationship to another object may give the other object *one* relationship back to the first object. One reference inside an object might be tied to one reference inside another object. This idea might apply to both loose items as well as list items.

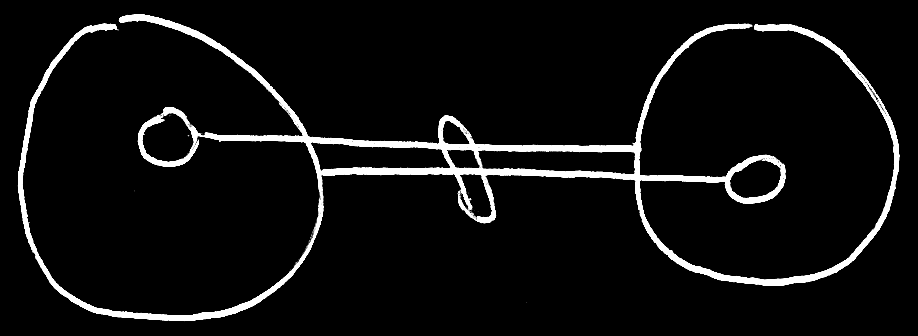
#### Diagram Notation

One idea about managing bidirectional relationships between objects might be: for each reference to an object, the other object might contain one reference back.

Drawing out the separate counterparts of a relationship between two objects might look like this:

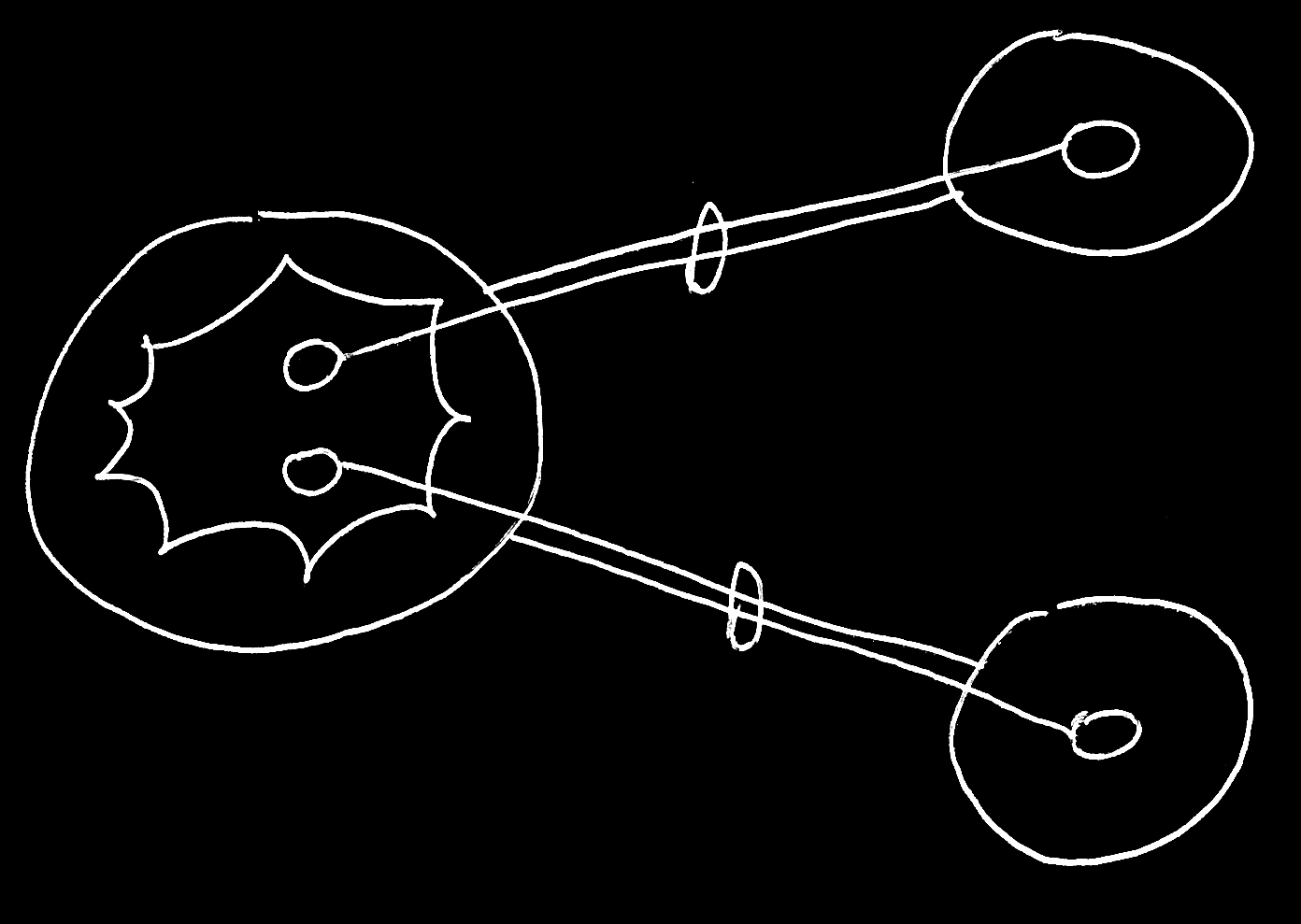


To express the closeness of the relationship between the two objects, a relational ring may be placed around the two lines:

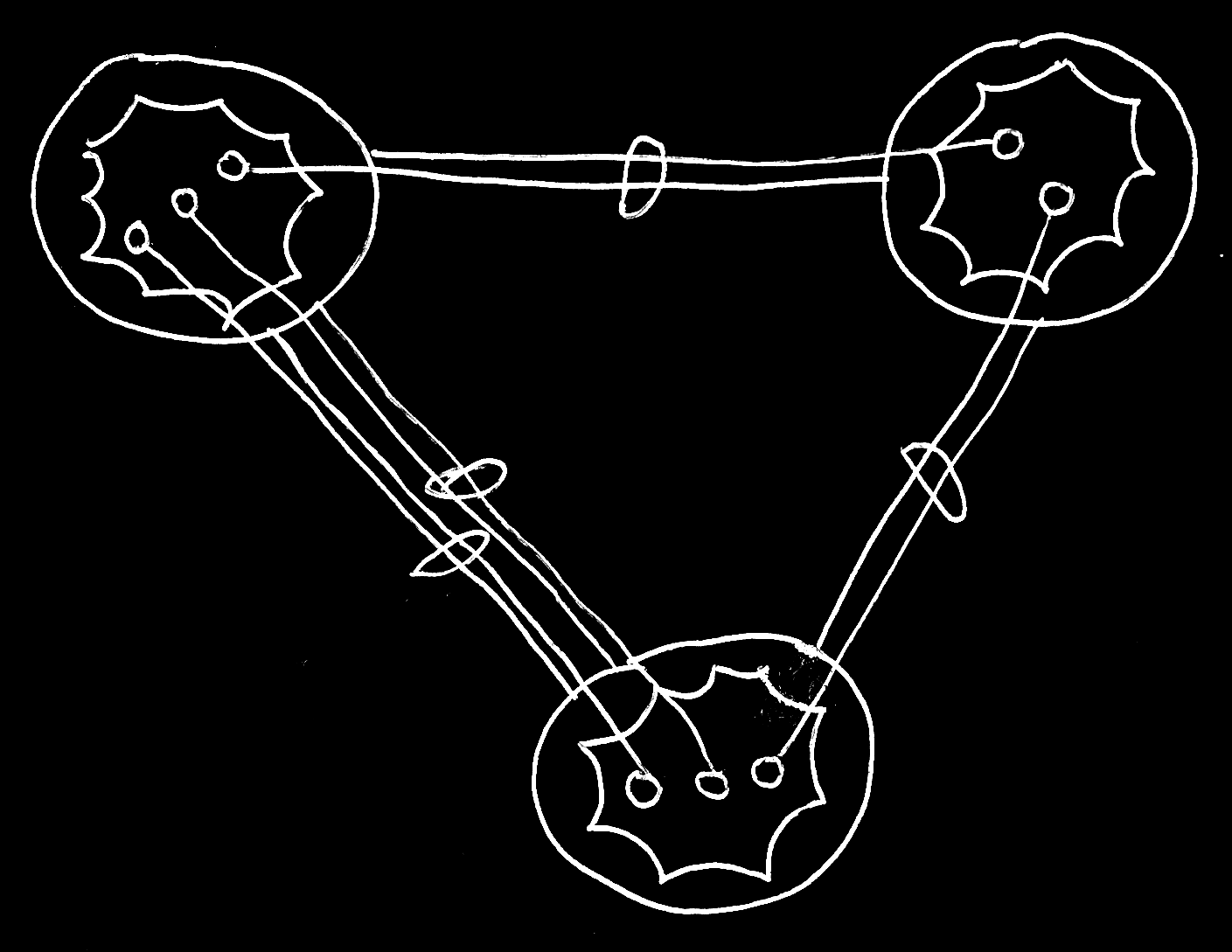


In 1 to n and n to n relationships the idea that each reference might contain one reference back seems to apply like this:

1 to n:



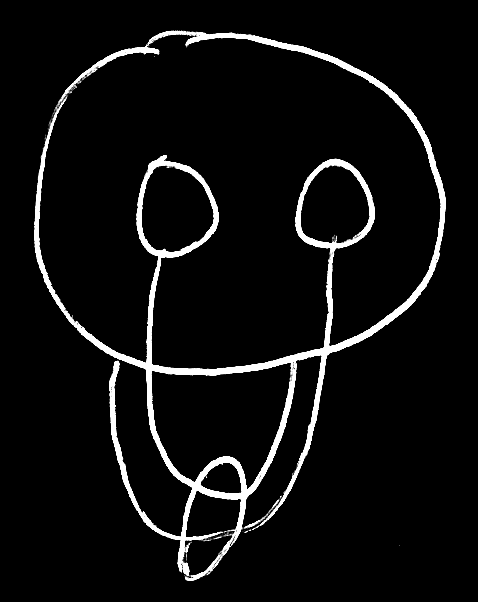
n to n:



It does not seem to matter whether an object reference is part of a list or not: every time it seems it is still *two* individual object references, that would be tied together.

#### Object relating to itself

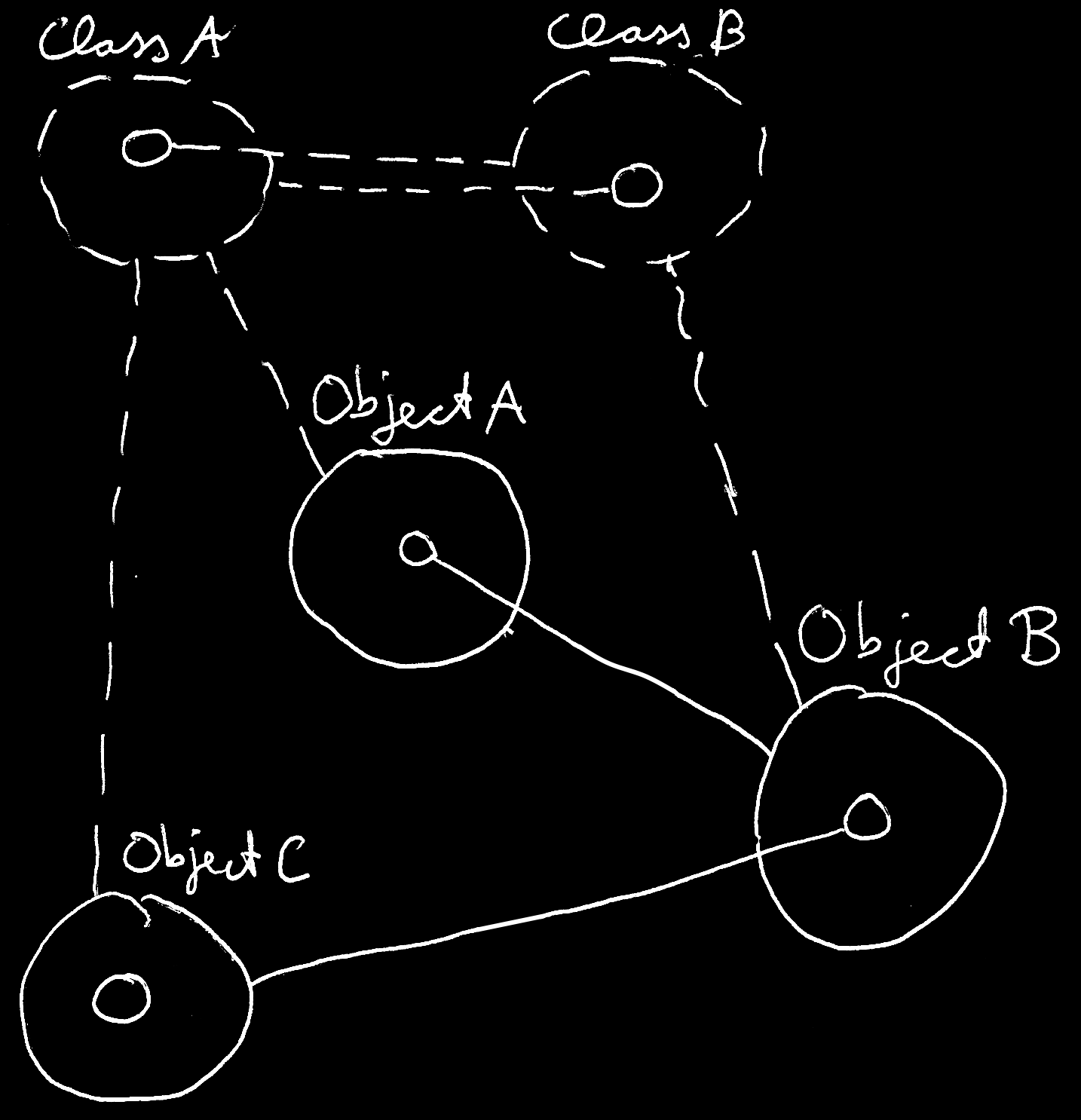
Sometimes an object might relate to itself. In a diagram this may look as follows:



### Bidirectional Relationship Synchronization

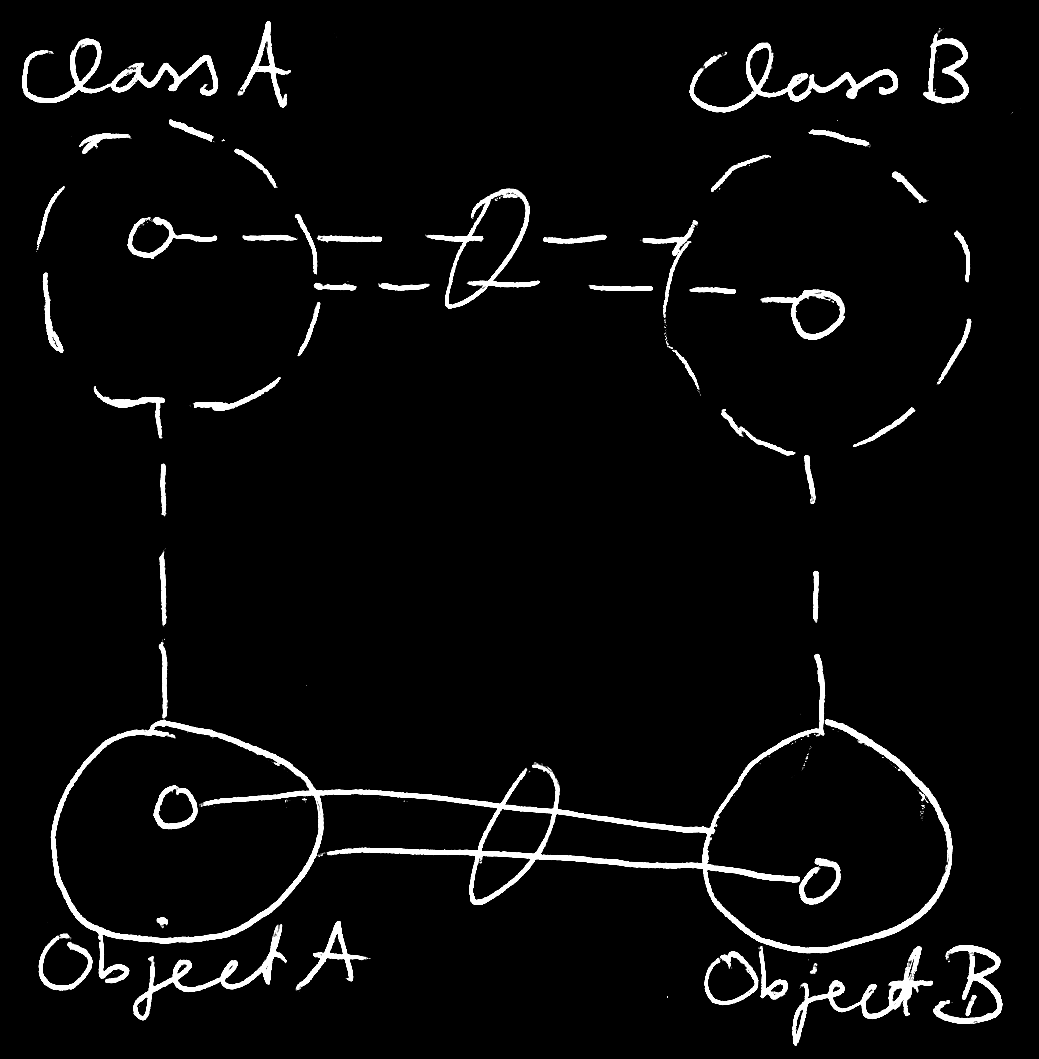
In a bidirectional relationship between classes, one class relates to another and the other class relates back to the first class. But that might not be enough to establish the relationship.

An object of one class might refer to an arbitrary object of another class, which then might refers back to an arbitrary object of the first class again, but not necessarily the object we started with.



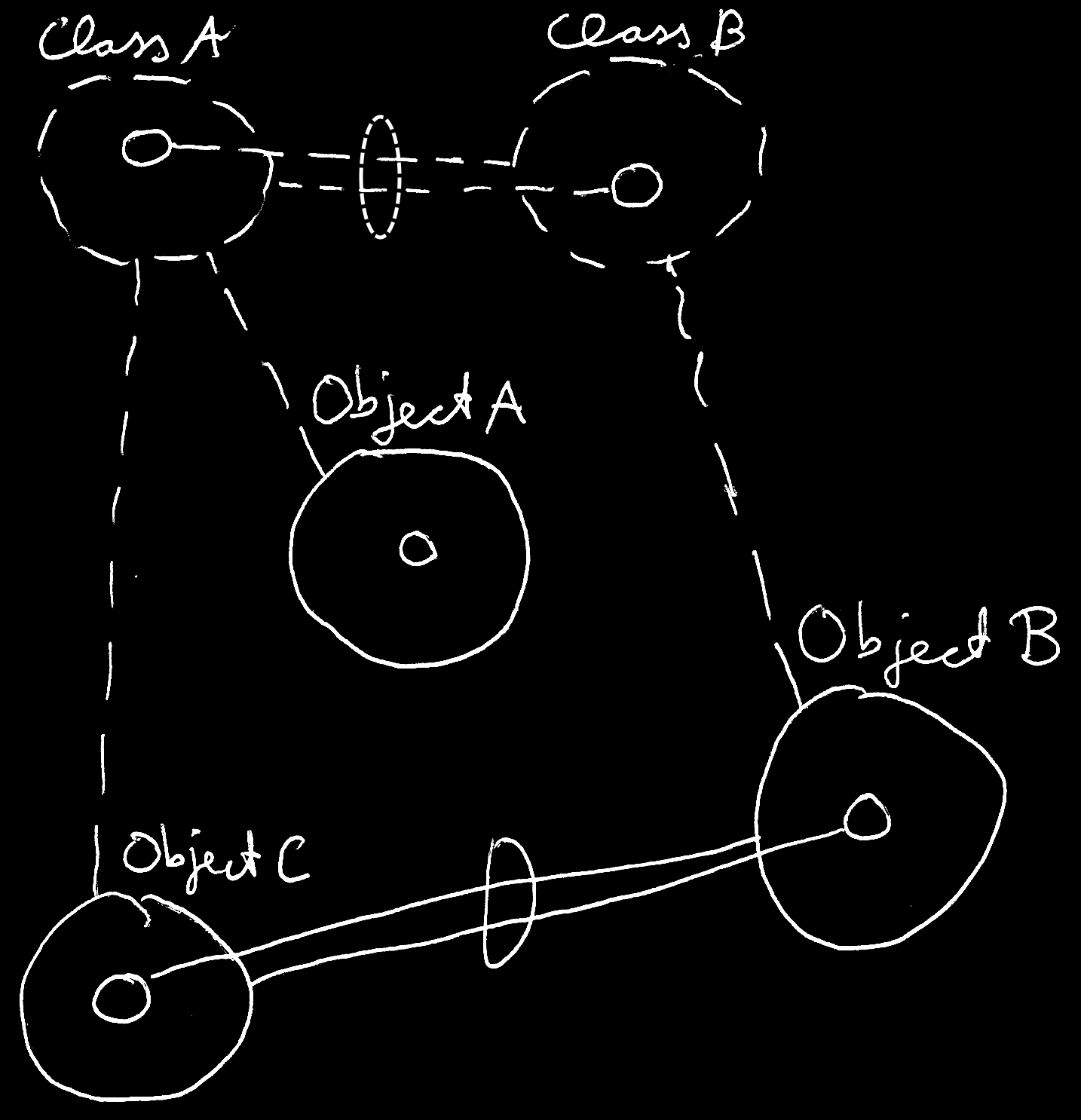
Two unidirectional relationships might not be synchronized with each other. An object of **Class A** may refer to an arbitrary object of **Class B**, which refers to an arbitrary object of **Class A** again. It might not make the two objects refer to each other. It would just makes the two objects refer to an arbitrary object of the other class, but not necessarily to each other.

To see to it one object relating to another would always make the other object relate back to the first object again, it may require the two counterparts of the relationship to be synchronized.



The tell-tale sign in a diagram, that a bidirectional relationship would be *synchronized*, could be a relational ring around lines that establish a bidirectional relationship.

When a sub-object's target would be changed, the other side of the relationship would also be updated.



Relationship synchronization would mean ensuring the integrity between the two counterparts of a relationship. For example, when a **Lid** is assignedto a **Jar**, the **Jar might** also assigned to the **Lid**.

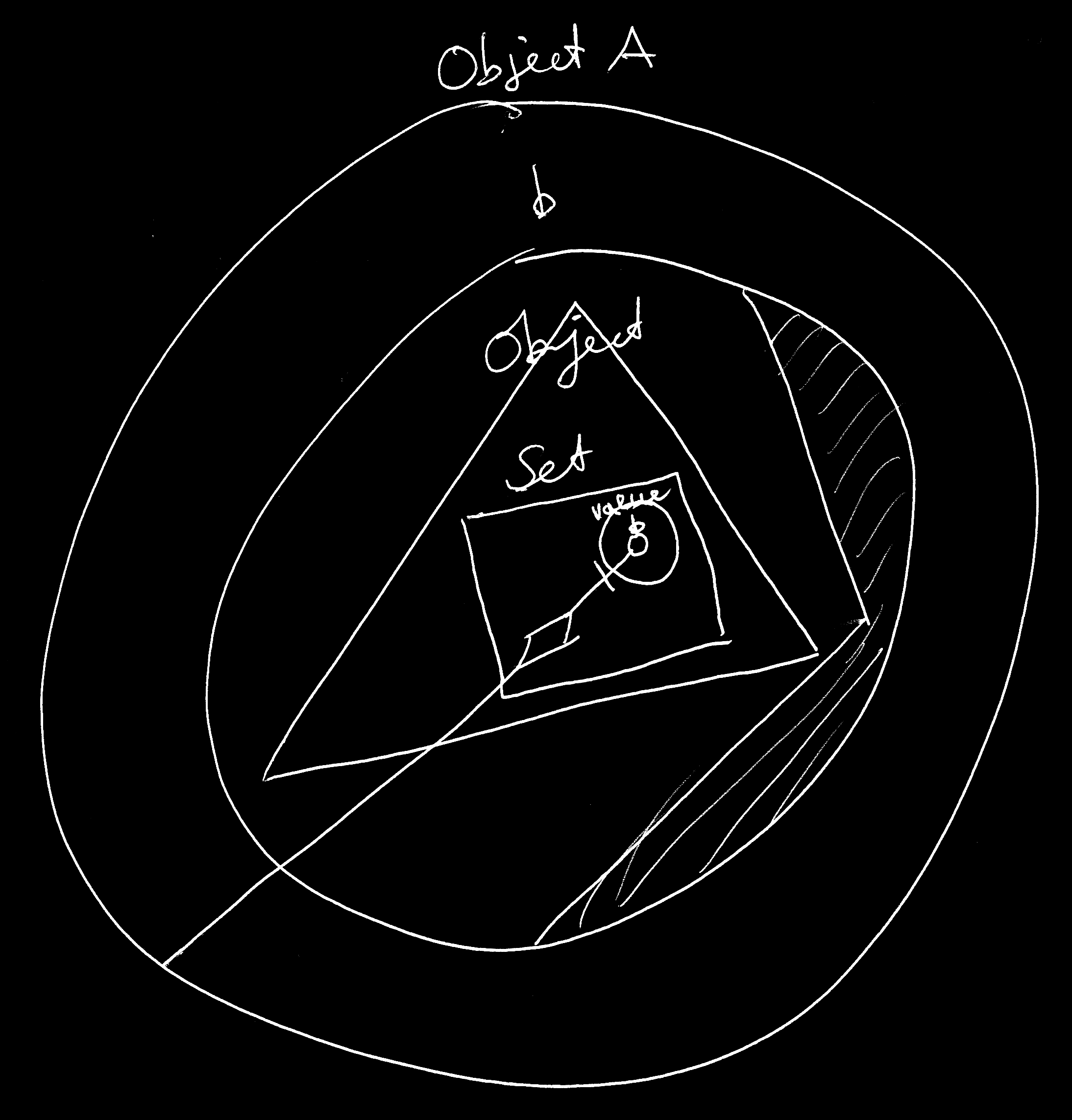
A complete bidirectional relationship might have three parts to it:

- One class has a sub-object of another class.

- The other class has a sub-object of the first class.

- The two unidirectional relationships are synchronized.

The idea is that software libraries could have code that make the relationship synchronization work. Circle Language Spec currently only aims to supply a notation for it. There should be freedom of choice how it is actually implemented. Procedures for relationship synchronization might be expressed in a diagram, when a system interface of a symbol is displayed.



The picture above is just an impression of what that might look like. There may be more than one way of doing it. System interfaces would be like the inner workings of symbols. Those inner workings might be tucked away and not always visible.

## Loose Ideas

### Loose Ideas about the\* Relationships Section

#### ± 2004

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

The\* classes are tied together with *relationships*. 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 Relationships collection. If\* you\* want to add a class, you\* do that in the\* Classes collection. If\* you\* want to add a relationship, you\* do that in the\* Relationships collection.

The\* Classes collection contains objects of class Class.

The\* Relationships collection contains objects of class Relationship.

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 relationships in the\* Structure.Relationships 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 **Relationship** consists of two **RelationClasses**. It contains two **RelationClass** objects that define the\* two classes of the\* relationship and how they relate to one another.

A **Relationship** also defines whether the\* relationship is **Bidirectional** or **Unidirectional**, by the\* **Boolean** **Bidirectional** member. If\* a **Relationship** is **Bidirectional**, then\* both classes are aware of eachother and refer to one another. If\* a **Relationship** is **Unidirectional** 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 **Bidirectional** relationship 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\* relationship 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 relationship classes within the\* relationship, but\* usually you\* won’t notice anything going wrong. So relax, but\* beware.

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

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

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

To define the\* quantity of x, you\* set **ExactNumber**. For instance, in a   
**Line n<=>2 Point** relationship, 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 relationship 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 relationship.

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

Structure

|

|-- Classes

| |

| |-- Class ()

| |

| |-- Attributes

| | |

| | |-- Attribute ()

| |

| |-- RelatedClasses

| |

| |-- RelationClass ()

|

|-- Relationships

|

|-- Relationship ()

|

|-- Bidirectional

|

|-- 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 **Relationships**.

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 **Relationships**.

### Loose Ideas about Bidirectional & Unidirectional

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

### Loose Ideas about Relationships 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-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 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 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

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

Relationships,

2008-11

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

JJ