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

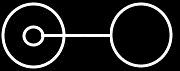
## Relationships

### 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 relations between classes compare to relations between individual objects.

### Relationships Between Classes

One object may relate to another object.

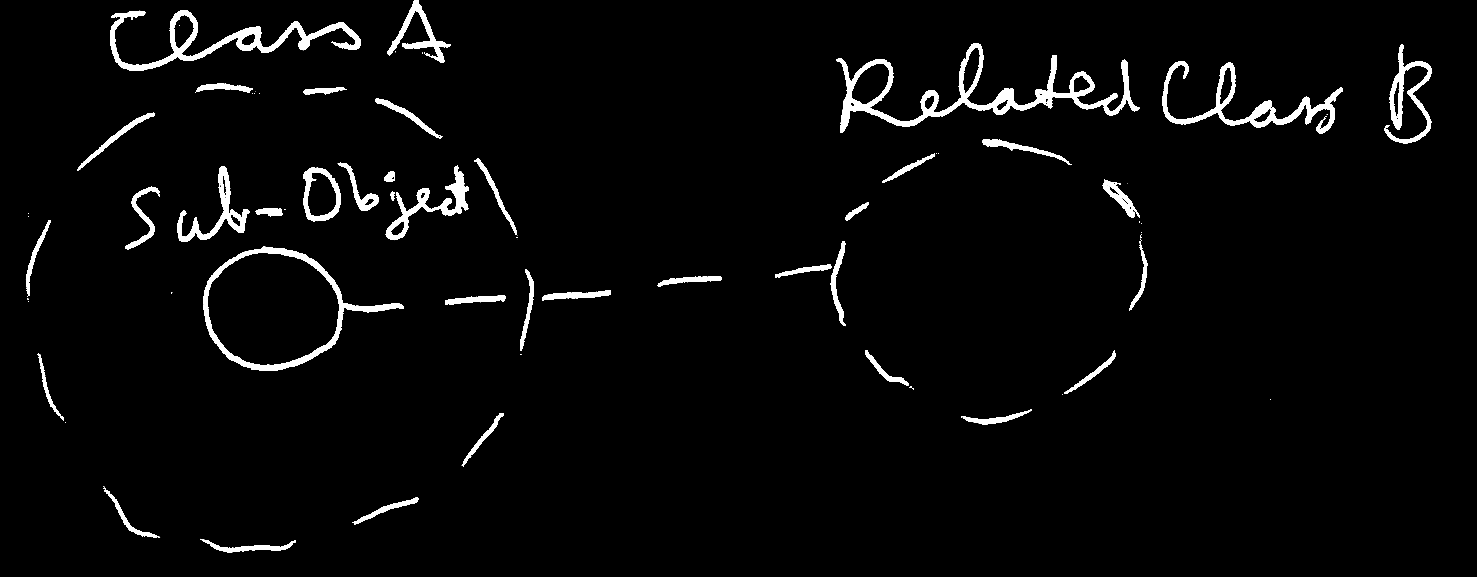


But that might not be what the topic of relationships commonly is about. It might be more about relationships between *classes* than individual objects.

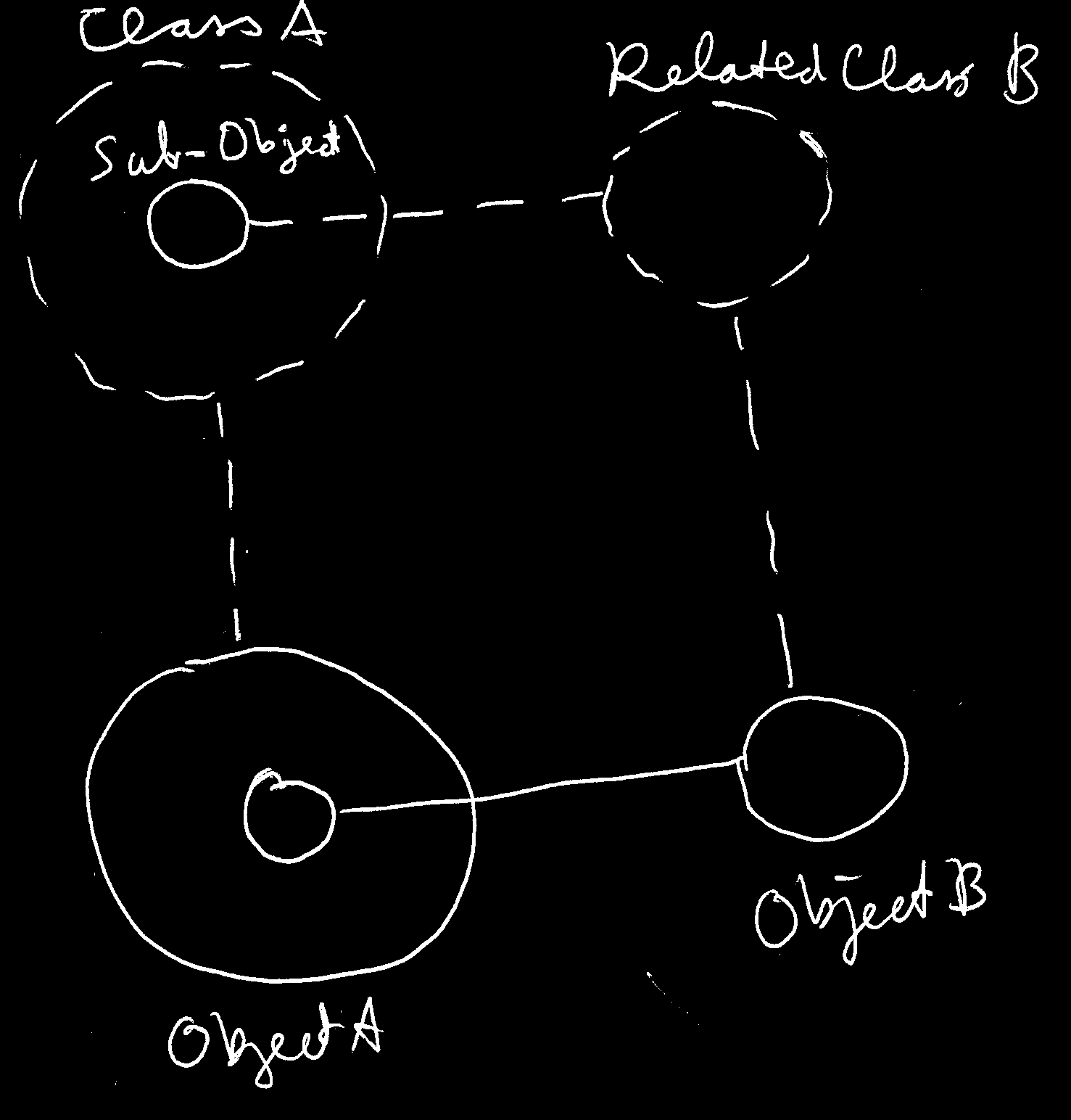


Relations between classes might determine the configuration of how objects are connected to each other, rather than just loosely tying together arbitrary objects.

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



When class sets the class for a sub-object like that, the sub-object might only ever be an object of that specific class.



**Object B** has a dashed line connected to **Related Class B** to point out its class. Otherwise, the connection would not be possible between the smaller circle inside Object A with a line pointing at Object B.

Here another example: an attempt to depict an object and its class with two more related classes.



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. (It may work differently, depending on different choices using dashed shapes.)

The class of a related item might not be set.



Then any type of object might be assigned as a related item. It would *not* introduce a relationship between classes.

A class might also specify related *lists*. Such a multiplicity of *many* may be expressed in a diagram with a nonagon symbol:



A nonagon might be placed inside a class, which would symbolize a class with a list inside of it:



When no class is assigned to the related list, it might imply that the related list could contain objects of any class. No relation between classes would be introduced by that.

When a class would be assigned 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 may create two relationships between 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:

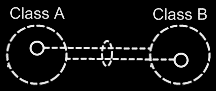


**Class A** has a sub-object of **Class B**. This would create a relationship from **Class A** to **Class B**.

**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:

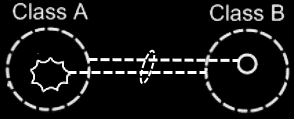


(The the relational ring's being dashed might be a stylistic choice. A solid line type or a slightly tilted ring shape might be variations.)

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:

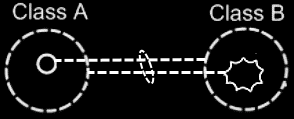


A nonagon would represent a list of things. Instead of letting a **Class A** contain a single item of **Class B**, it may contain a list of items of **Class B**:

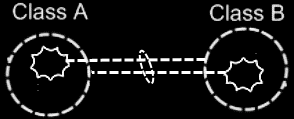


The picture above aims to express an **n => 1** relationship between **Class A** and **Class B**.

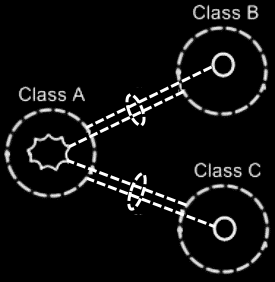
The picture below would display a **1 => n** relationship between **Class A** and **Class B**.



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:

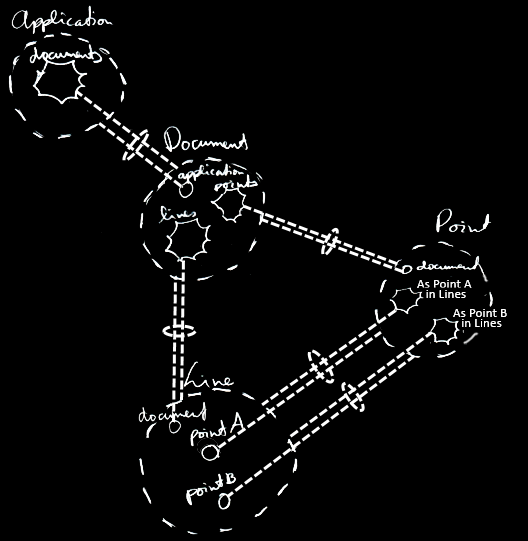


#### Example

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* might set guidelines for how objects could connect to eachother.

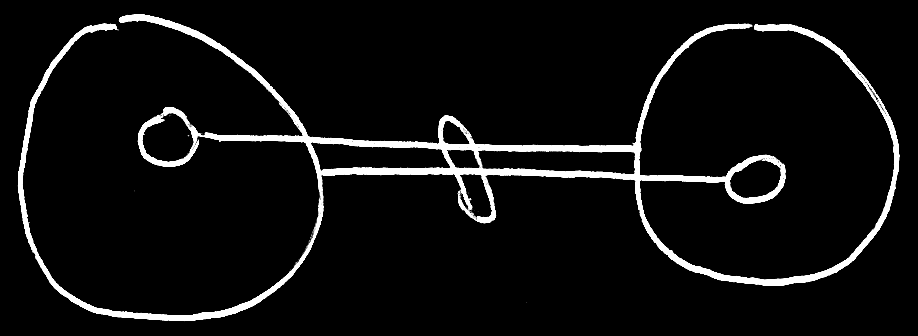
Relationships between *objects* would bethe *actual* connections between objects.

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:

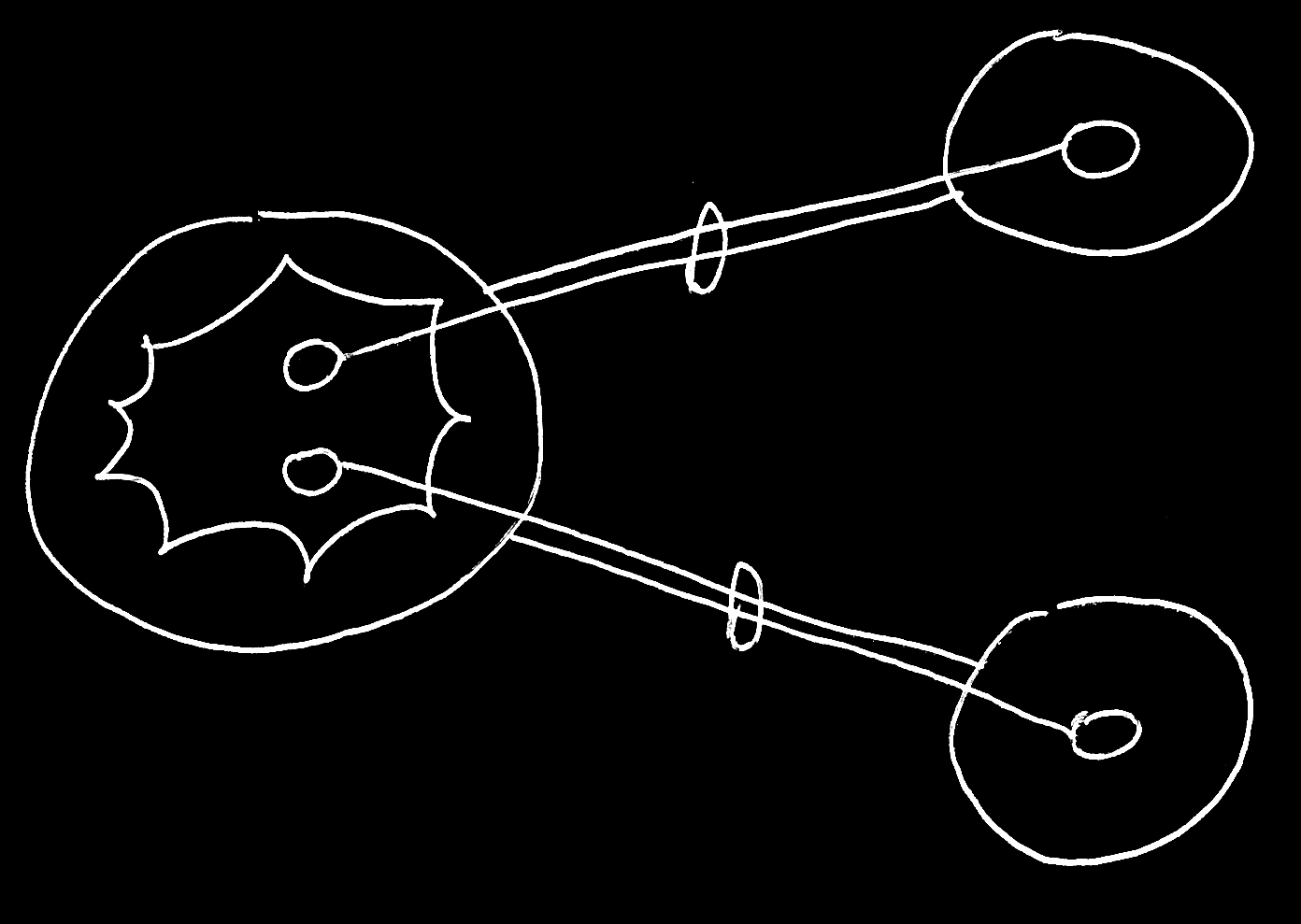


To express the closeness of the relationship between the two objects, a relational ring may be placed around the two lines:



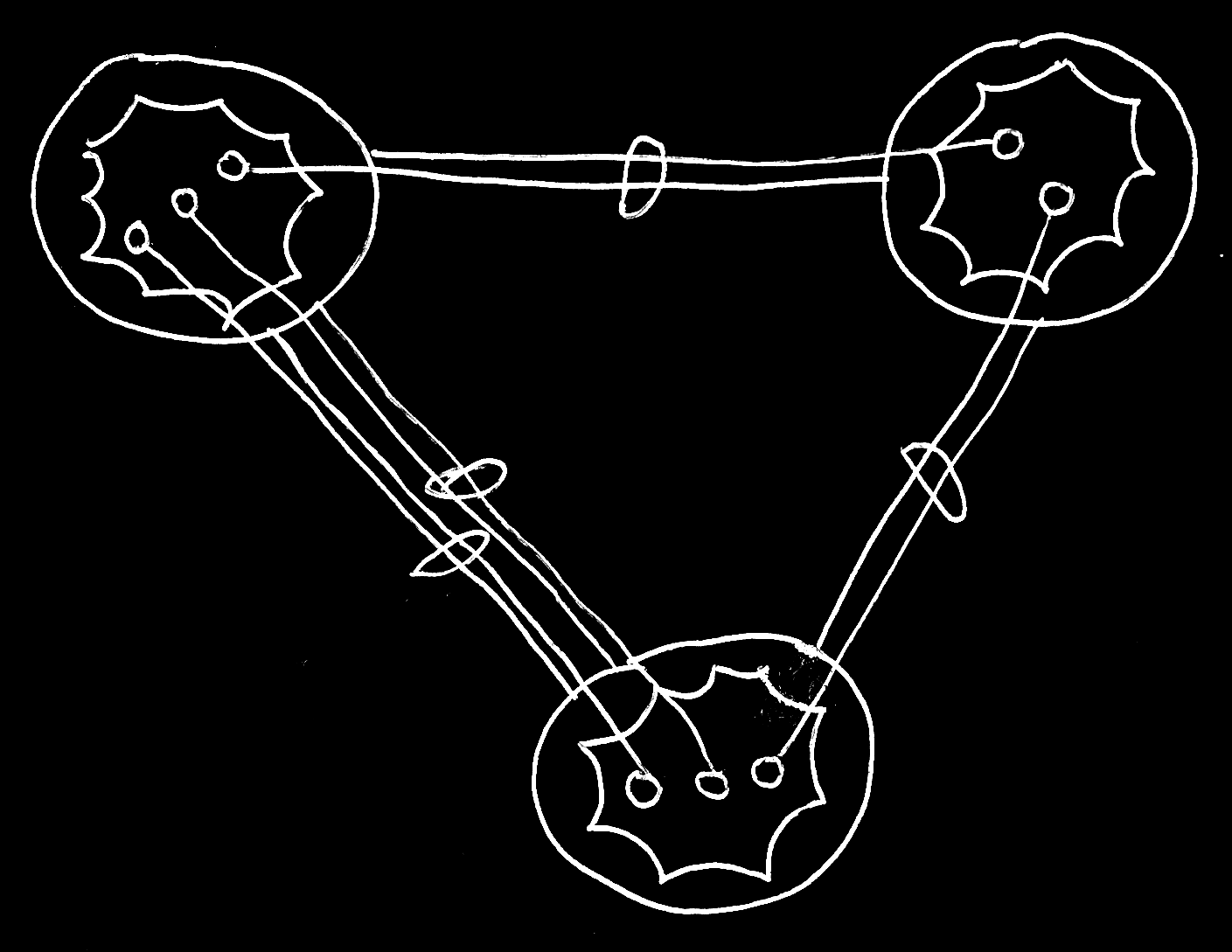
The bidirectional relationships between objects all seem be **1** to **1.** When one object would refer to another, the other one would refer back to the first one. **1** to **n** and **n** to **n** relationships between classes might result in multiple **1**to**1** relationships between objects.

1 to n:



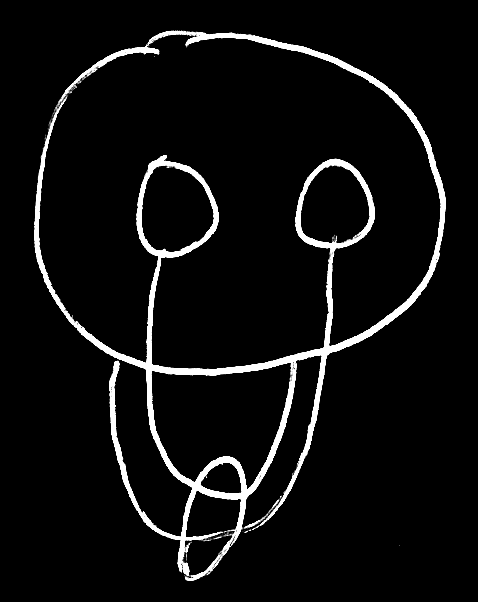
A multiplicity of **n** may create lists inside classes. In objects of those classes these lists might contain separate items. Any item in the list references an object and that may give the other object *one* reference back again.

n to n:



#### Object relating to itself

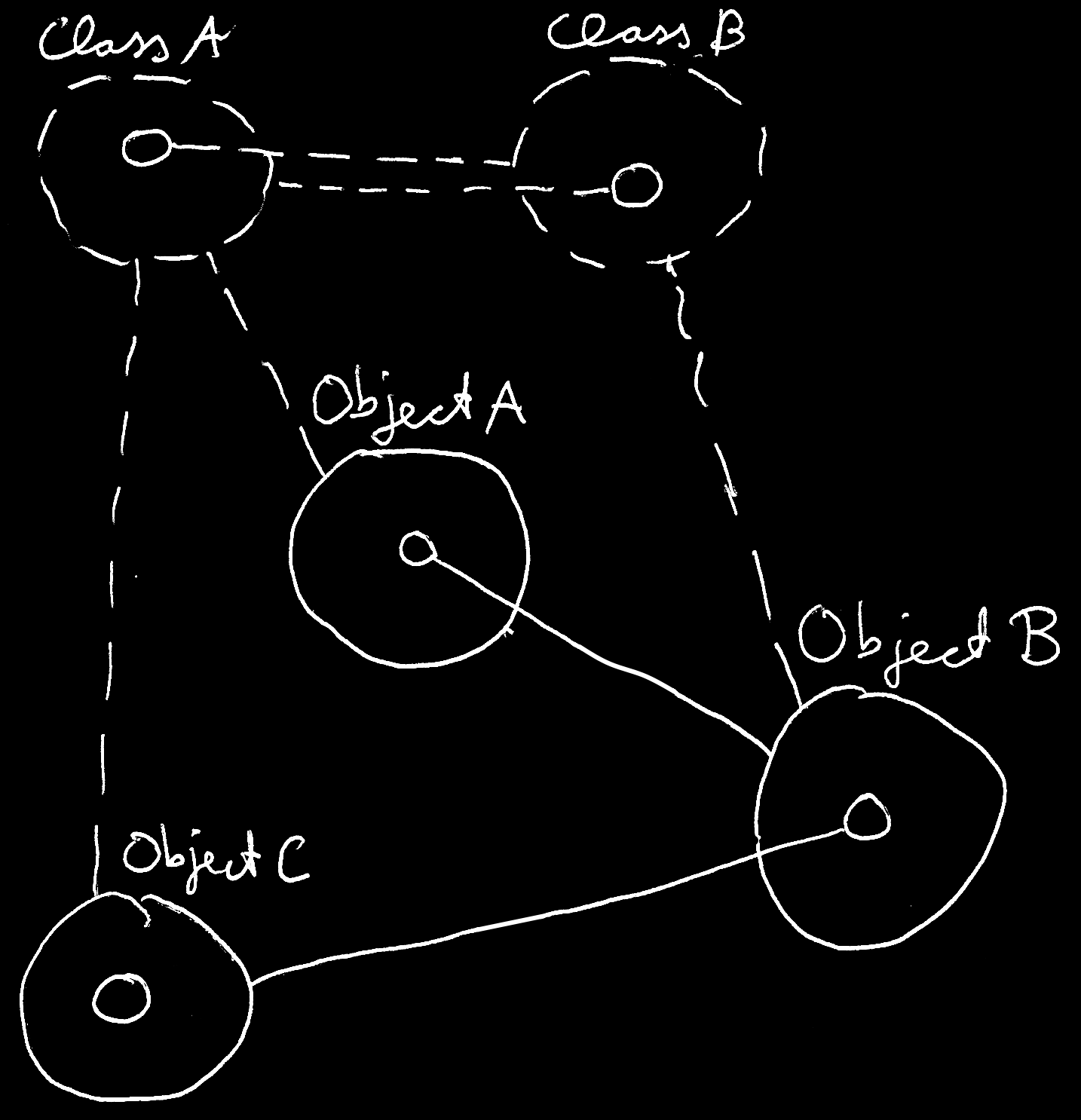
Sometimes an object might relate to itself. In a diagram this may look as follows:



### Bidirectional Relationship Synchronization

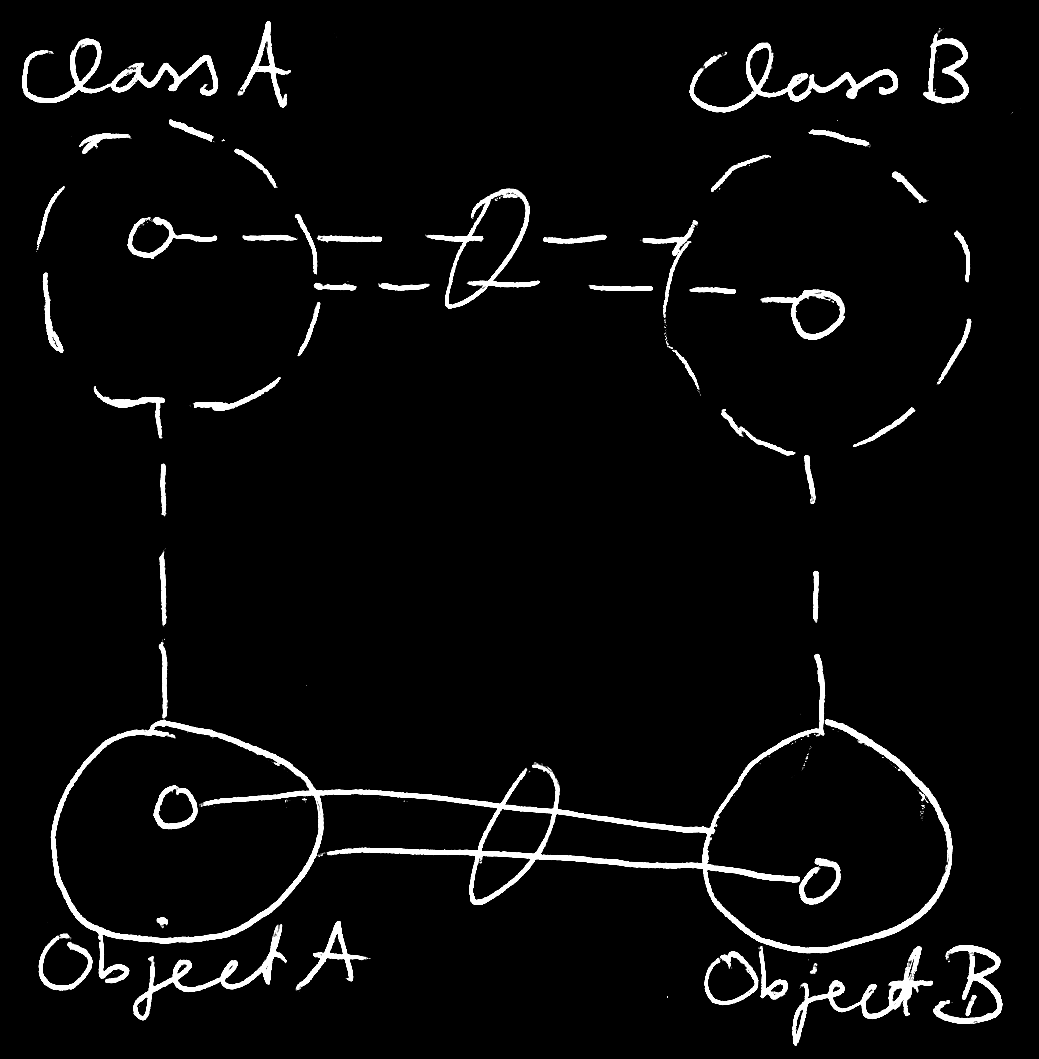
In a bidirectional relationship between classes, one class relates to another and the other class relates back to the first class. But that might not be enough to establish the relationship.

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.



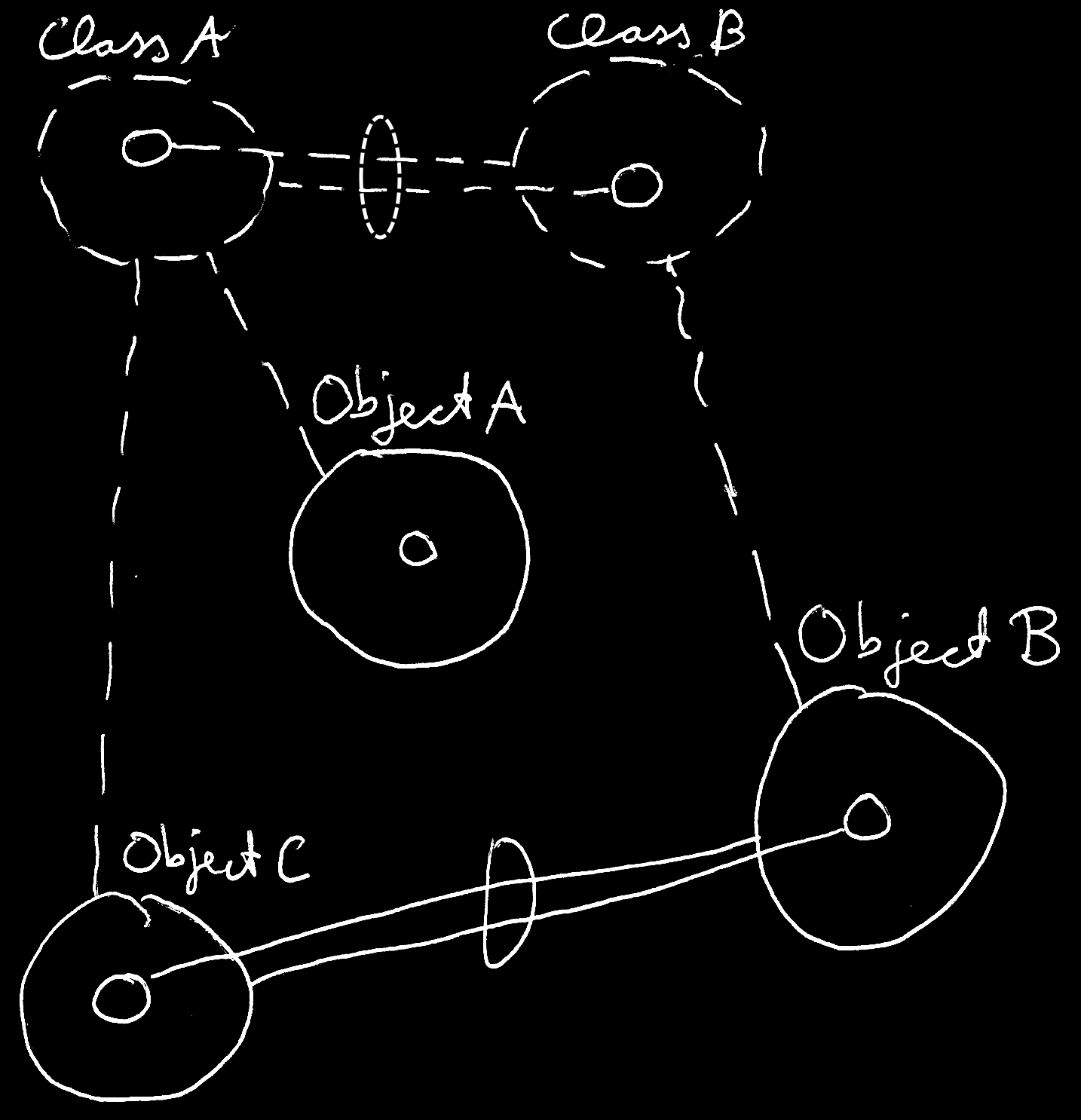
Two unidirectional relationships might not be synchronized with each other. An object of **Class A** may refer to an arbitrary object of **Class B**, which refers to an arbitrary object of **Class A** again. It might not make the two objects refer to each other. It would just makes the two objects refer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

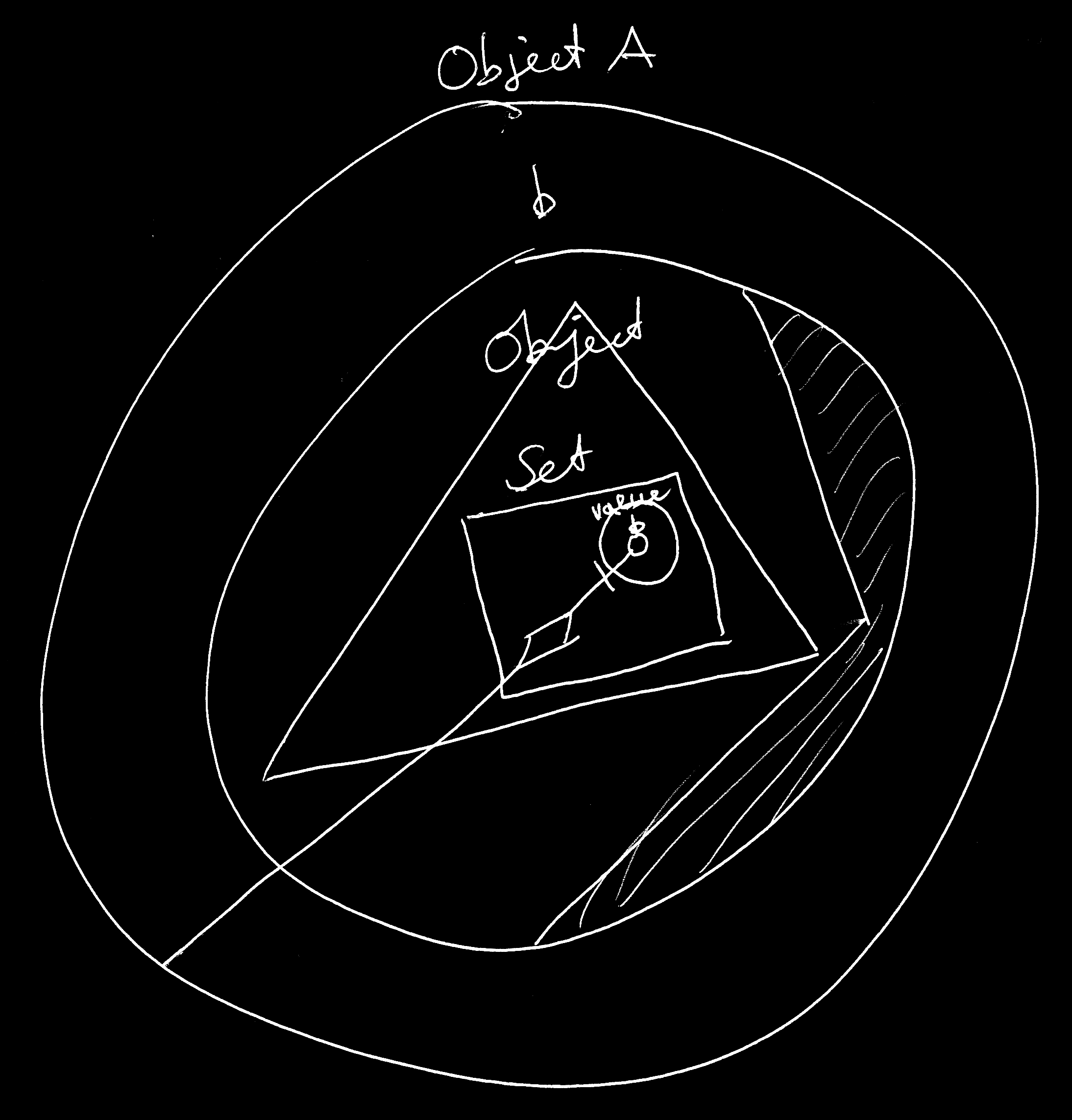
A complete bidirectional relationship might have three parts to it:

- 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.