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

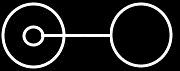
## Relationships

### Introduction

This topic aims to introduce the idea 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. An attempt is also made to describe how relations between classes would compare to relations between individual objects.

### Relationships Between Classes

One object may relate to another object.



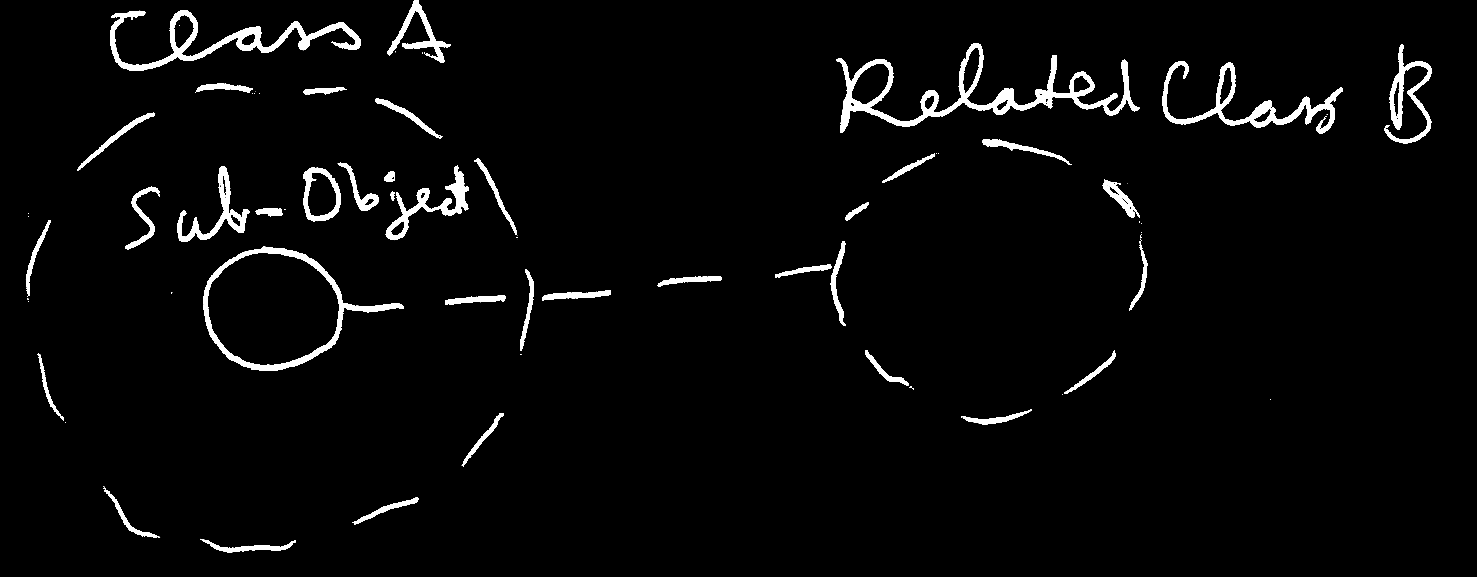
But that might not be what the topic of relationships would commonly be about. It might be more about relationships between *classes* than individual objects.



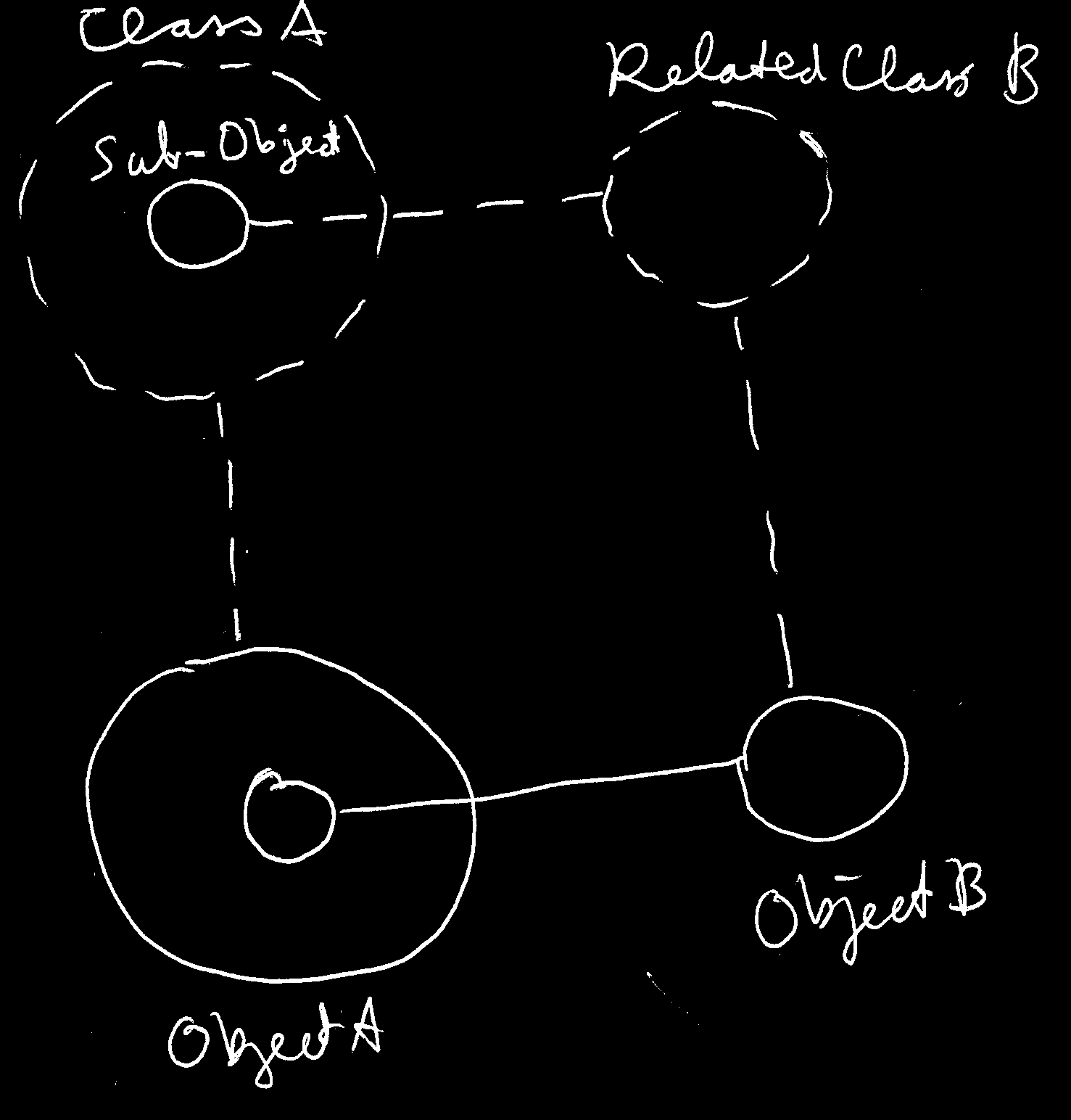
Relations between classes might determine the configuration of how objects are connected to each other, rather than just loosely tying together arbitrary objects.

#### Related Item with a Class

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



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



**Object B** would point out a class with a dashed line connected to **Related Class B**. Otherwise, **Object B**'s connection with the solid line might not be possible.

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 lines and shapes.

#### Related Item Without a Class

The class of a related item might not be set.



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

#### Related Lists

A class may also specify related *lists*. That might be expressed in a diagram with a nonagon symbol:



#### Related List Without a Class

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



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

#### Related List with a Class

When a class would be assigned to the related list, it may suggest the related list might only contain items of this class.



#### Related List with Multiple Classes

There is also the idea that a related list might be assigned *multiple* classes, which could 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

#### Compared to Unidirectional Relationships

The relationships described so far would be *unidirectional*: one way only. A unidirectional relationship between one class and another might look like this:



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

Relationships might also be *bidirectional:* a two-way street. If one class relates to another, then the other class might relate back to the first class again.

#### 1 to 1 Relationship

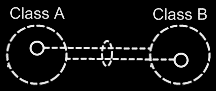
**Class B** does not seem to have a relationship back to **Class A** yet. The picture below would add that relationship back to **Class A**:



In case of a bidirectional relationship if one class gets a sub-object of another class, the other class might also get a sub-object pointing back.

#### Relational Ring

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 picture above would express a **1 to 1** relationship between **Class A** and **Class B**. (The relational ring's being dashed might just be a stylistic choice.)

#### Lists

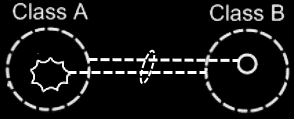
Other multiplicities might also be used. A multiplicity of **n** might be expressed with a nonagon:



A nonagon might represent a list of things.

#### 1 to N Relationship

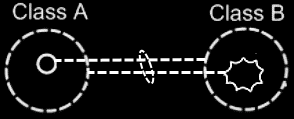
There might be **1 to n** relationships between classes. Instead of letting a **Class A** contain a single item of **Class B**, it may contain a list of items of **Class B**:



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

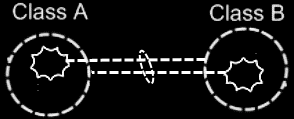
#### N to 1 Relationship

The picture below aims to display a **n to 1** relationship between **Class A** and **Class B**.



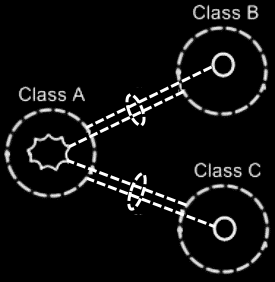
#### N to N Relationship

There may also be **n to 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. The picture below aims to display an **n to n** relationship between **Class A** and **Class B**.



#### Relationship with Multiple Classes

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



#### Class Relating to Itself

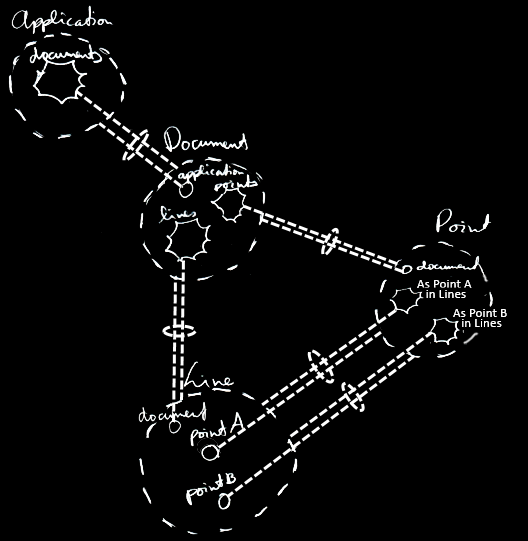
A class may relate to itself. For instance, a person might relate to a parent, which could also be a person. So then a person would be related to a person, which might relate the person class to itself.

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**.



The example attempts to display all the classes, relationships, related items and related lists of the class structure. Here would be 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** could have a **1 to n** relationship with the **Document class**. Viewed in the opposite direction, this could make a **Document** reference the single **Application** it would be opened in. A **Document** might hold a collection of **Points** and a collection of **Lines**. That would make **Document** have a **1 to n** relationship with **Point** and a **1 to n** relationship with **Line**. Inversely, this would make a **Point** or **Line** point outone **Document** it belongs to. Furthermore, a **Line** could be composed of two **Points**: **Point A** and **Point B**. The idea is that P**oints** could be reused in multiple lines. Perhaps a bit creatively, these became collections inside a **Point** 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 to n** relationship, involving lists of lines.

### Bidirectional Relationships Between Objects

Relationships between *classes* might set guidelines for how objects could connect to each other. Relationships between *objects* could bethe *actual* connections between objects.

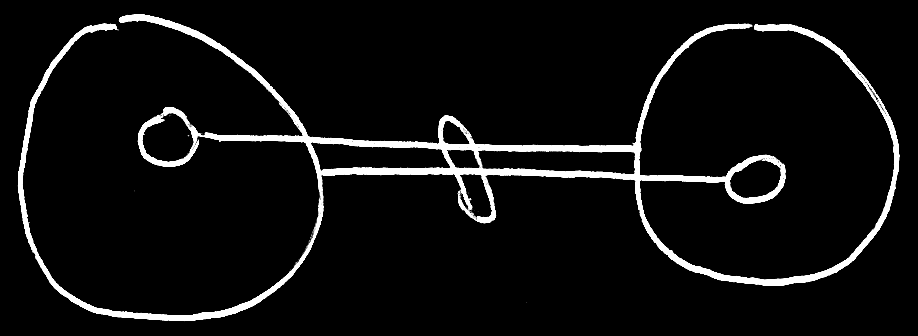
#### Pairs of Related Objects

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:

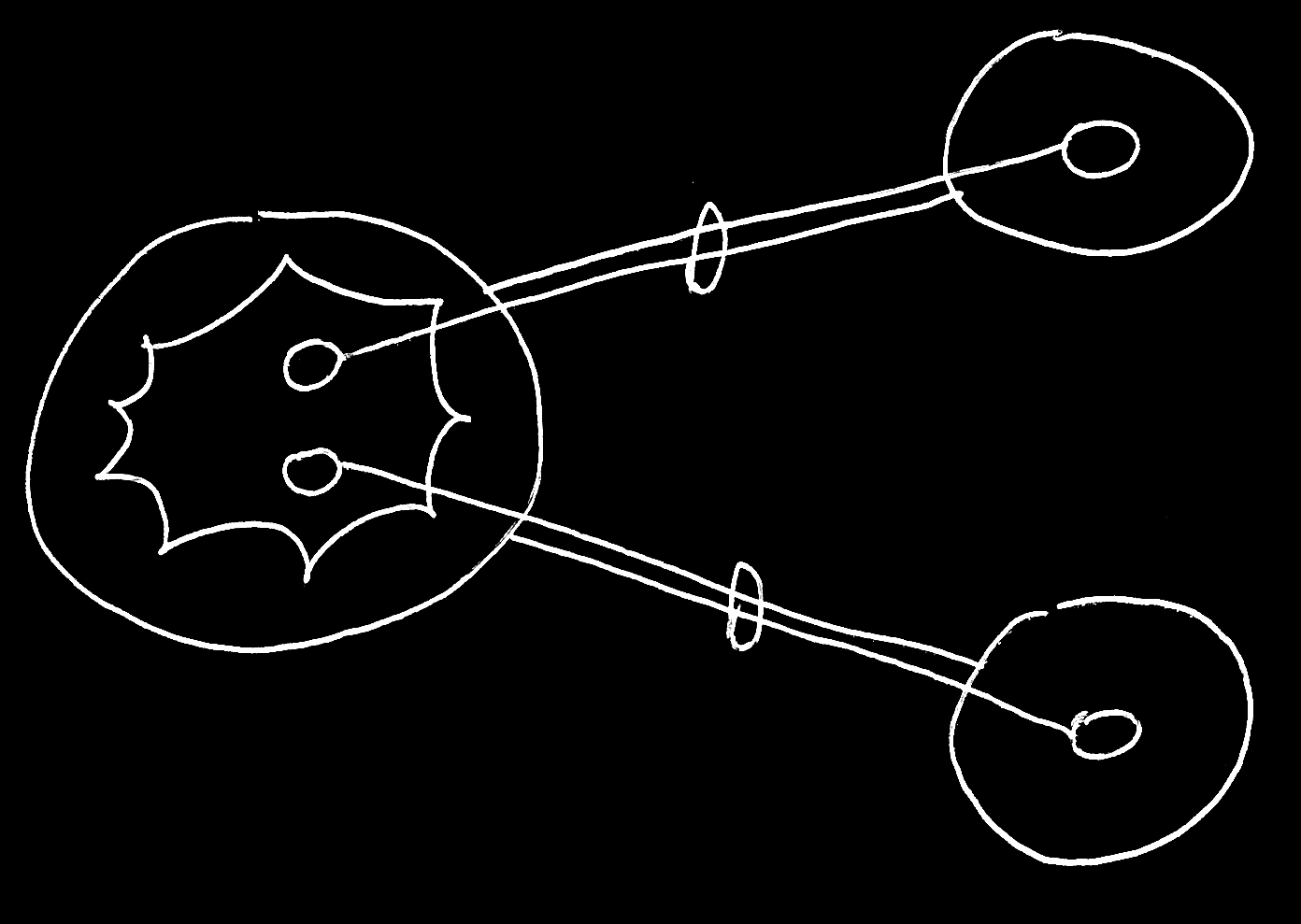


The bidirectional relationships between objects all seem be **1** to **1.** That might be a bit of a bold statement. And it might be just one way of looking at it. But here the idea is entertained that when one object would refer to another, the other one would refer back to the first one.

#### Lists

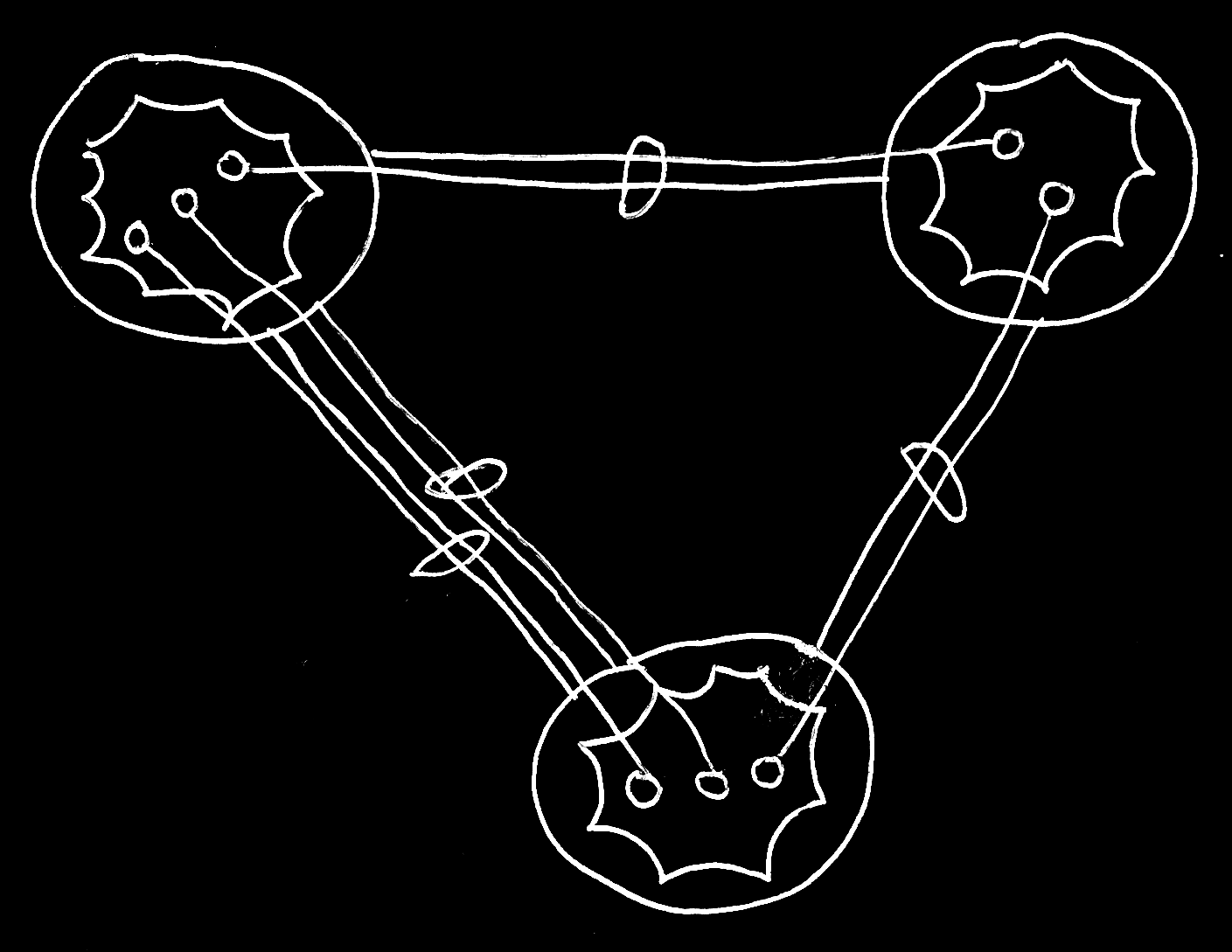
**1** to **n** and **n** to **n** relationships between classes might result in multiple **1**to**1** relationships between objects.

1 to n:



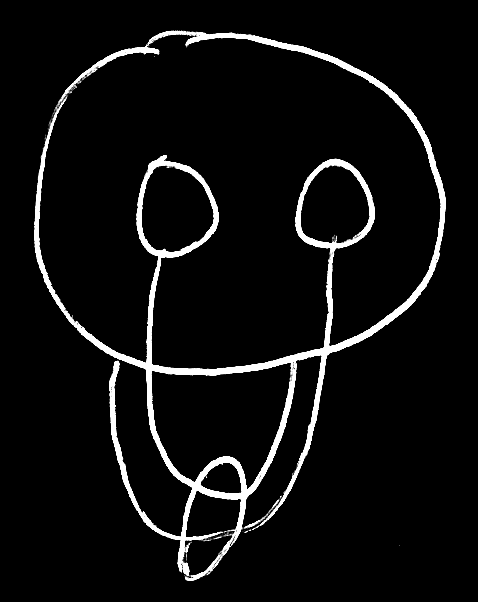
A multiplicity of **n** may create lists inside classes. In objects these lists might contain separate items. Any item in the list references an object and that might give the other object *one* reference back again.

n to n:



#### Object Related to Itself

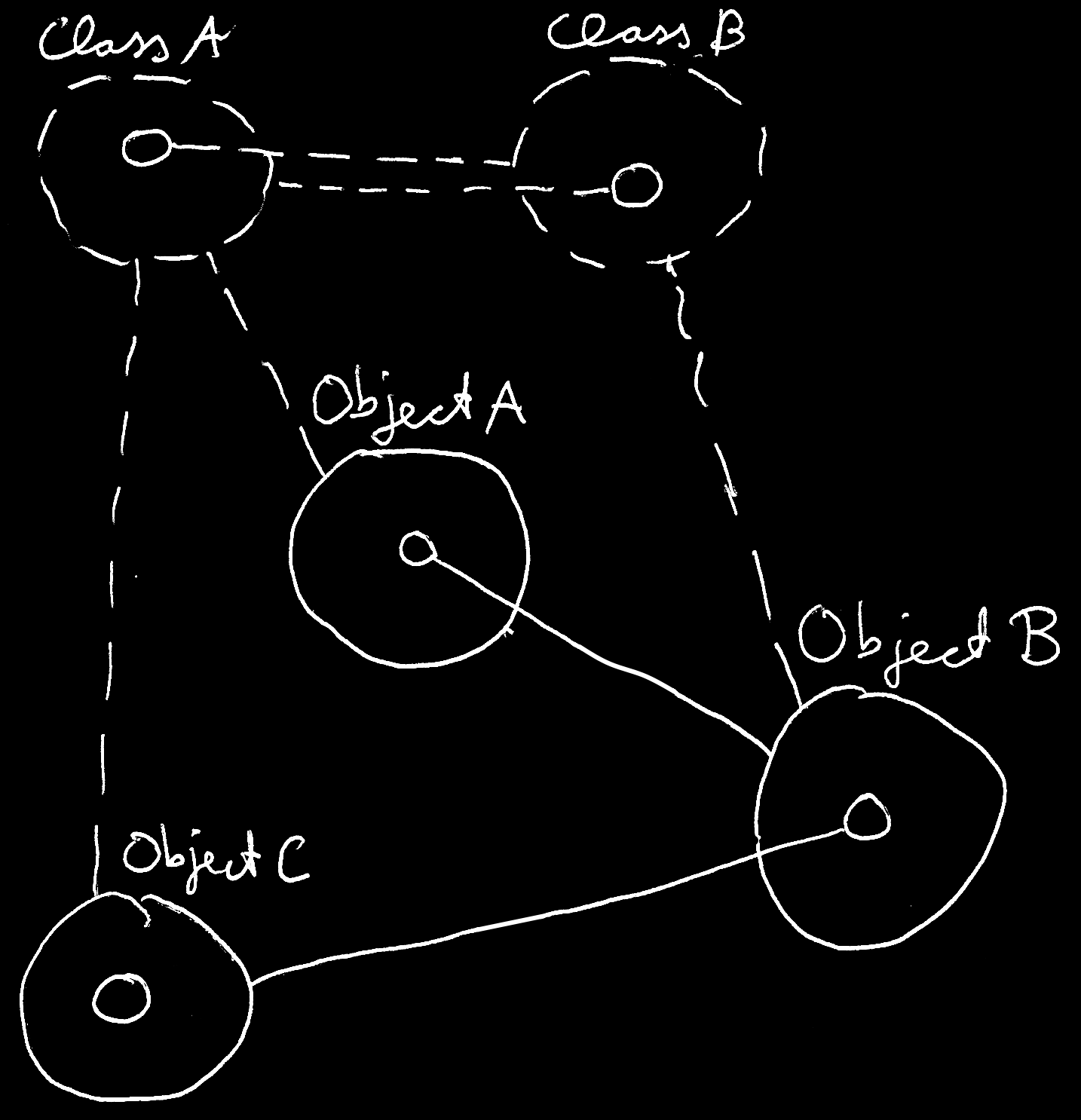
Sometimes an object might relate to itself. In a diagram this may look as follows:



### Bidirectional Relationship Synchronization

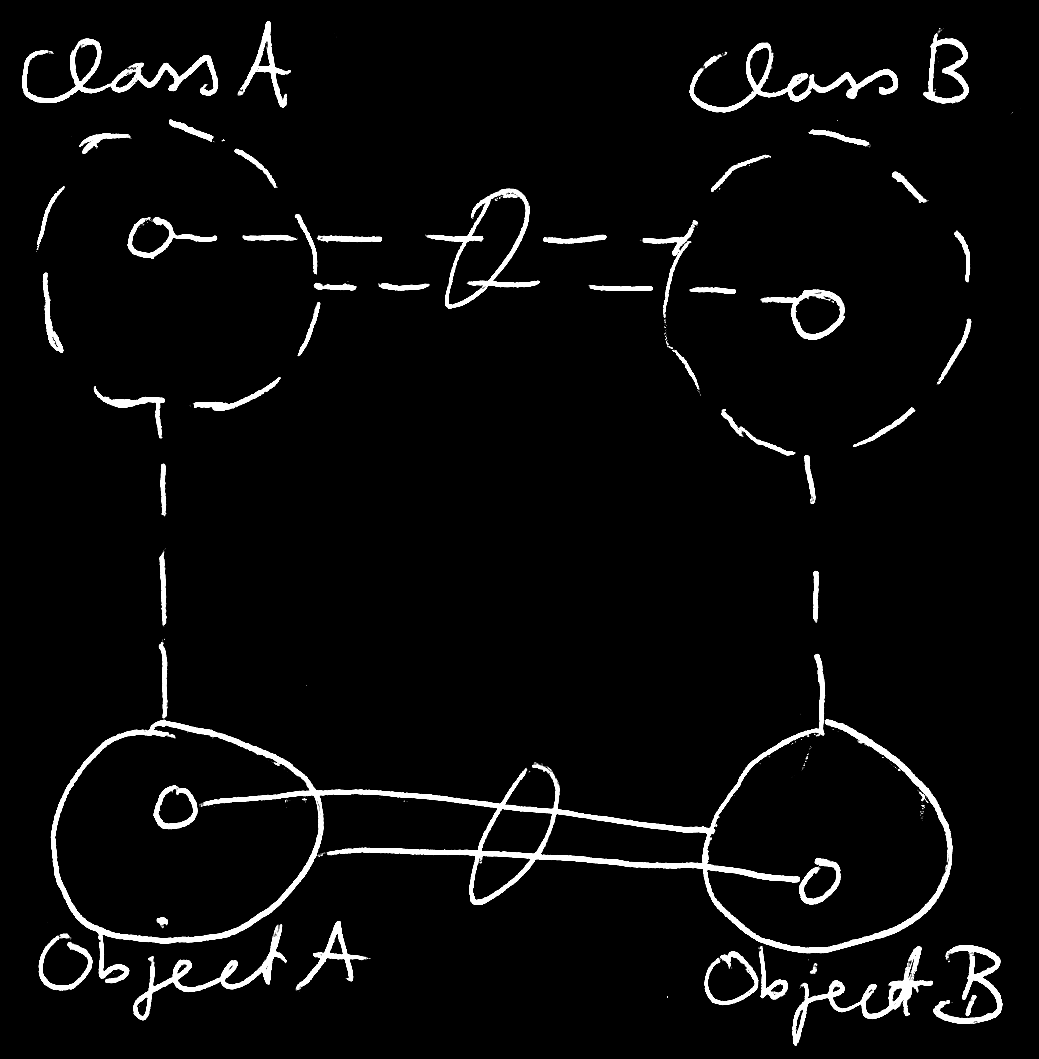
In a bidirectional relationship between classes, one class would relate to another, and the other class would relate back to the first class. But that might not be enough.

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



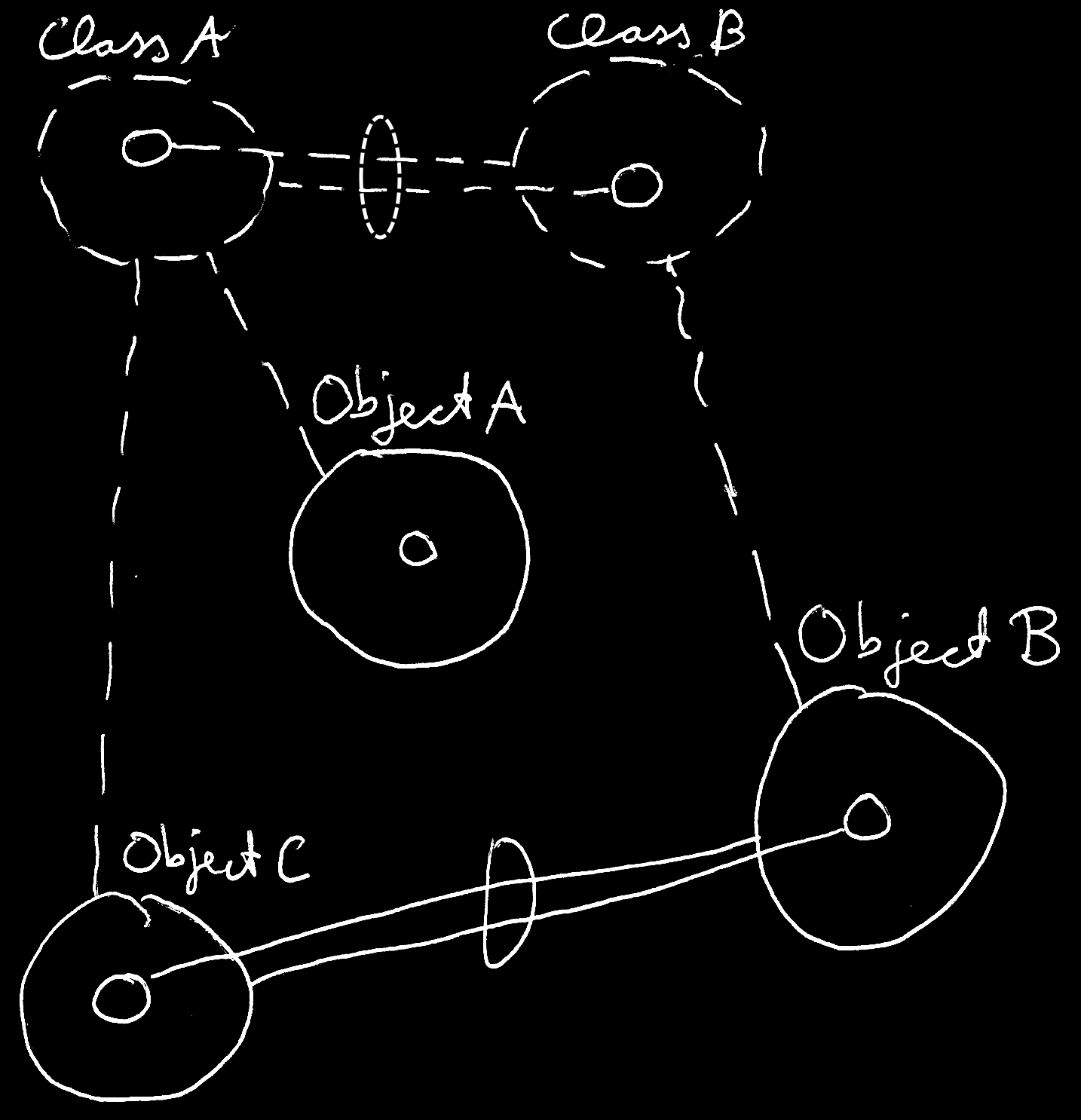
Two unidirectional relationships might not be synchronized with each other. It might not make the two objects refer to each other. It would just make the two objects refer to an arbitrary object of the other class.

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



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

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



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

As such, a bidirectional relationship might have three parts:

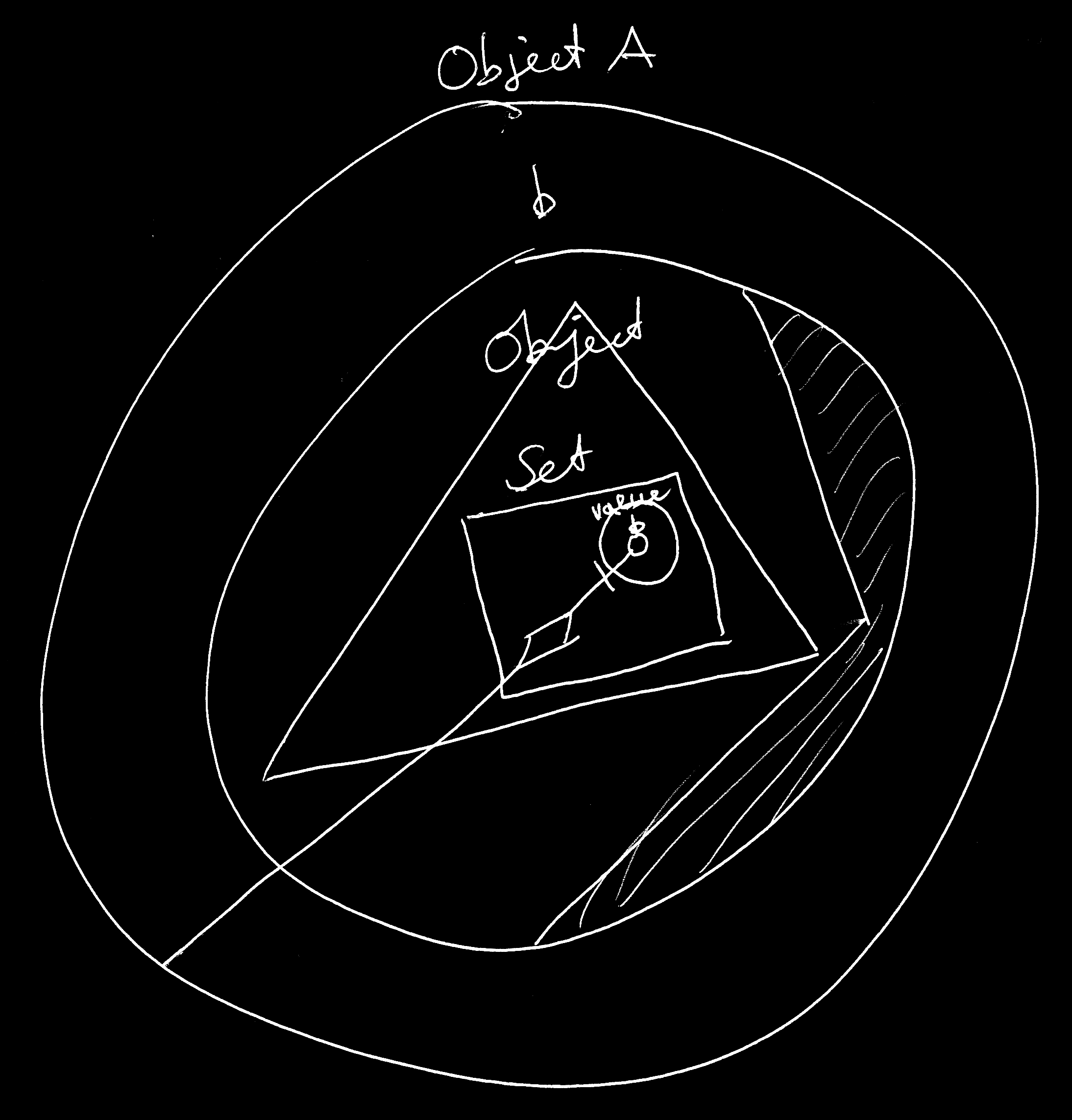
- One class has a sub-object of another class.

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

- The two unidirectional relationships may be synchronized.

The idea would be that that there might be software libraries that 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 would be implemented. The notational choices made here, and implementation details might not always resonate with each other. Variations on notations may be possible. The one described here might lay a foundation.

The implementation procedures for relationship synchronization might be expressed in a diagram, when a *system interface* of a symbol would be displayed.



System interfaces would be like the inner workings of a symbol. Those inner workings might be tucked away and not always visible. System interfaces may have a chapter on their own. The picture above would just be an impression of what that might look like. There might be more than one way of doing it.