|  |
| --- |
| Circle Language Spec: System Objects |

## System Objects

### Introduction

All systems are composed of *system objects*. The main purpose of system objects is to manage relations between objects. System objects are *actual* objects, that all other systems are based on. Most of the system objects are part of the *code base*.

System objects control *system aspects*, such as **Class**, **Execution** and **Values**. Aspects as such are controlled through system commands. Assignment commands are also system commands. An assignment command copies an aspect from one object to another.

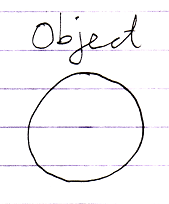
The *System Objects* documentation covers connectors, connections and all the different notational forms that come with controlling system aspects.

### System Objects

This article demonstrates the system objects that all systems are composed of. The main purpose of system objects is to manage relations between objects.

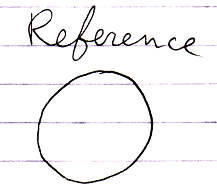
#### The Object

The most basic element of a computer program is the *object*. An object represents thing, an idea or a place, a number or a collection or anything else. All those things are called objects.



#### The Reference

You always deal with *references* to objects.



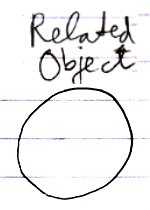
An object is never directly accessed.

An object *reference* is an *undecorated* pointer. It is used by other system

objects, that wrap an object reference into a context.

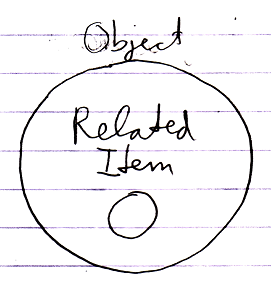
#### Related Object

A synonym for *object reference* is *related object*, except that a related object is in the context of another object.

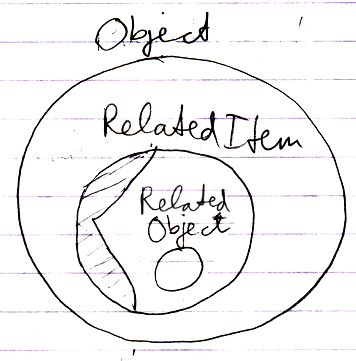


#### Related Item

A parent object can contain *related items*.



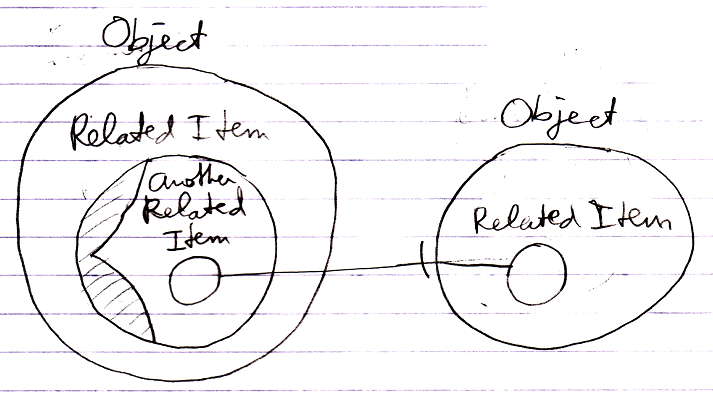
It is the result of a relation. A related item wraps a related object.



A related object is the undecorated object pointer wrapped inside a related item.

#### Pointer-to-Pointer

A related item can also wrap yet another related item, contained by another parent object.

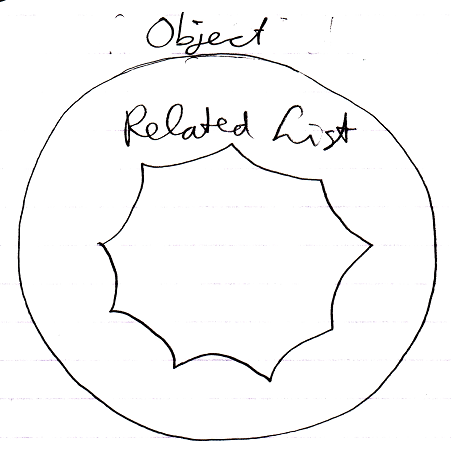


In that case it is said to be a *pointer-to-pointer*. The use of pointers to pointers makes you able to let something else determine what is eventually targeted.

You never work with objects directly, so even though the diagram above looks like a pointer, it is really a pointer-to-pointer.

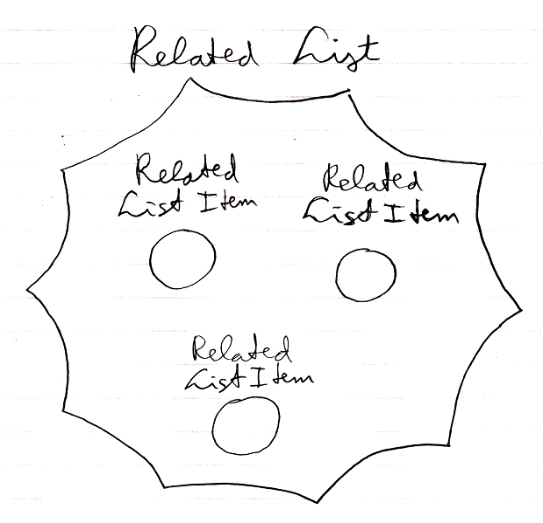
#### Related List

A parent object can also contain *related lists*.



#### Related List Item

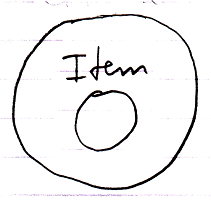
A related list can contains multiple *related list items*.



A *related list item* is much like a *related item*. Often, when the term related item is used, it also applies to related list items. In some cases though, the behavior is different.

#### Item

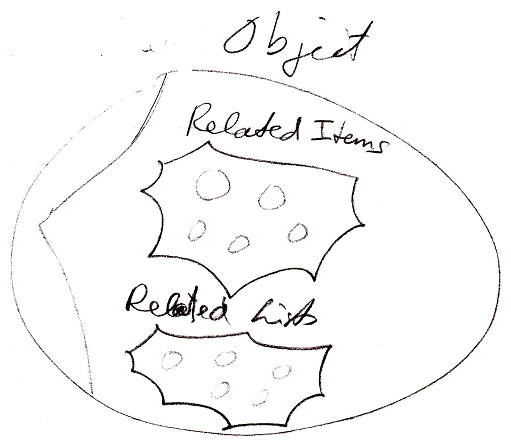
Another name for an object is an item. But when we speak of an item, we are usually think about an item inside a certain *context*.



So an item is a related object.

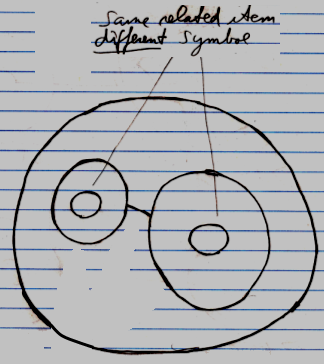
#### Related Items & Related Lists Collections

An object’s related items and related lists are actually stored as the object’s having two collections: **Related Items** and **Related Lists**. The **Related Items** collection contains all the related items of the object. The **Related Lists** collection contains all the related lists of the object.



#### Symbol

The term *symbol* should not to be confused with the term related item. A symbol is a shape displayed on screen. The same related item can be displayed on screen multiple times.



A symbol is not a system element, it is just a shape displayed on screen. *Symbol* is a diagram expression concept, not a coding concept.

However, sometimes the term symbol is used as a synonym for related item, because it just talks a lot easier. But they are not *really* synonyms.

#### System Object

All the terms above indicate different kinds of *system objects*, except for the term *symbol*.

#### Summary

Most of the system objects are part of the *code base*. System objects without any extensions need to be implemented right inside the code base. *Extensions* can be added later using the new computer language itself. System objects are *actual* objects, that all other systems are based on.

A recap of all the terms introduced:

**Object**

**Reference**

**Related Object**

**Related Item**

**Pointer-to-Pointer**

**Related List**

**Related List Item**

**Item**

**Related Items & Related Lists Collections**

**Symbol**

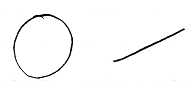
**System Object**

### System Aspects

The behavior of objects, references and lists is controlled by controlling their *aspects*. This article lists out the various aspects of objects, references and lists. The pictures show the aspects’ main diagram symbolization.

#### Object

The **Object** aspect determines which object is pointed to.



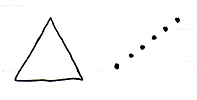
#### Class

The **Class** aspect determines which other object will function as the prototype of another object. It can also bind a reference to a class.



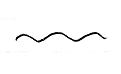
#### Interface

The **Interface** aspect allows you to control how objects look on the outside, while the insides of the objects can differ completely. The **Interface** aspect is separately covered in the *Interfaces* article group, and will not be mentioned in the *System Aspects* article group any further.



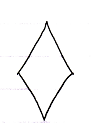
#### Value

The **Value** aspect allows you to store binary content and allows you to yield over values from one object to another.



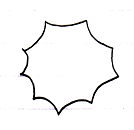
#### Execute

The **Execute** aspect is about being able to execute an object as a command.



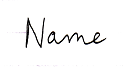
#### List

The **List** aspect allows you to add and removeitems from a list.



#### Name

The **Name** aspect allows you to give names to objects, lists and references.



#### Existence

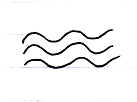
The existence aspect allows you to create a new object. You can also annul an object reference to make it point to nothing. You can also *check* whether an object reference is **Nothing**.



(This is the main symbolization of **Nothing**.)

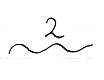
#### Data

With the data aspect you can control whether you have reading or writing access. Instead of being able to access-control the direct object, the **Data** aspect controls read-write access to all of the sub-objects as well.



#### Clone

The **Clone** aspect is related to the **Value** aspect, but will also copy values of sub-objects.



#### Reference

Sometimes no aspect of a reference is called upon, but there is worked directly with the reference itself. That is not really an aspect, but in that case it is said you are calling upon the **Reference** aspect.



#### Summary

Here is the list of aspects again:

**Object**

**Class**

**Interface**

**Value**

**Execute**

**List**

**Name**

**Existance**

**Data**

**Clone**

**Reference**

### Object-Bound & Reference Bound Aspects

When controlling system aspects, it is important to realize that some aspects are bound to an object, while other aspects are bound to a reference.

Object-bound aspects are controlled by an object. The object also controls its sub-objects’ reference-bound aspects. Therefore *reference-bound* aspects can also be called *sub-object-bound* aspects.

The following aspects are object-bound:

**Value**

**Class**

**Execute**

**Clone**

**Data**

**List**

The following aspects are reference-bound (or *sub-object-bound*):

**Reference**

**Object**

**Existence**

**Class**

**Name**

The **Class** aspect is both object-bound as well as reference-bound.

Understanding that some aspects are object-bound is important, for instance because it makes sure that object-bound aspects are protected by the object and not all of a sudden become unprotected through another reference.

An object controls its object-bound aspects as well as its sub-objects’ reference-bound aspects.

### System Commands

System *commands* are the commands of system objects, through which aspects of objects are controlled.

Most aspects have a **Get** and a **Set** command, but each aspect is controlled in a different way.

#### System Commands for the Reference Aspect

The only system command for the **Reference** aspect is:

**Reference Get**

A reference can be a **Related Item** or a **Related List Item**. This creates two overloads for **Reference Get**:

**Reference Get 🡪 Related Item Get**

**Reference Get 🡪 Related List Item Get**

The reference aspect is used in pointer operations.

#### System Commands for the Object Aspect

The main two system commands for the **Object** aspect are:

**Object Get**

**Object Set**

**Object Get** retrieves the targeted object of a reference.

**Object Set** changes an object reference target.

Those cover all the standard situations, but pointer-to-pointer situations make things more complex.

##### Pointer-to-Pointer

In a standard situation **Object Get** and **Object Set** control references to objects. However, a reference can also point to yet again another reference: to a related item contained by another parent object. This makes the other parent object decide which object is eventually pointed at.

As such, pointer-to-pointer functionality introduces extra commands.

##### Set Object to Reference

To be able to set the object aspect to another related item, **Object Set** has two overloads:

**Object Set** 🡪 **Set Object to Other Related Item**

**Object Set** 🡪 **Set Object to Other Related List Item**

If you want a single name to express both situations, you could call it **Set Object to Reference**.

##### Get Object which is a Reference

Because the object aspect can be another related item, the **Object Get** command gets two overloads:

**Object Get** 🡪 **Get Object which is Another Related Item**

**Object Get** 🡪 **Get Object which is Another Related List Item**

If you want a single name to express both situations, you could also call it **Get Object which is a Reference**. During execution these system commands call **Reference Get** on the other related item.

##### Use Reference As Object

The reference aspect can be access-controlled for the different ways you can use it. Pointer-to-pointer situations require you to be able to use a reference as an *object*. To be able to access control the different purposes for which you can use a reference, the **Reference Get** command gets the second implementation:

**Use Reference As Object**

which delegates directly to the **Reference Get** command.

##### The Overloads Recapitulated

Do not wreck your brain over all this delegation and overloading. It is just for pointer-to-pointer situations to have the same command names as standard situations, and also to be able to separately access control the specific *uses* of a reference. You will not usually see the pointer-related commands, because they will be implicitly delegated to by the main system commands. This leaves us with the following commands:

**Object Get**

**Object Set**

**Use Reference As Object**

Detail: For that last command you might want to overload **Object Get.** But that does not work. You can not overload it, because they will both take a pointer to an object as an argument. To disambiguate, they have to have a different name and you have to point to a *specific* command.

#### System Commands for the Class Aspect

The main system commands to control the **Class** aspect are:

**Use As Class**

**Class Set**

**Reference-Class Get**

**Object-Class Get**

**Use As Class** is like **Object Get**, but then for the purpose of using that object as the class of another object. This is common usage of the class aspect. It also makes you able to separately access control whether an object can be used as a class.

The **Class Set** command is executed on an object reference. The reference can then only point to objects of that class. **Class Set** applies only to references, and not to objects, because the class of an object is only set upon creation.

The commands **Reference-Class Get** and **Object-Class Get** getthe class object that is associated with a reference or object. Those commands are actually less commonly used.

##### Class is both Object-Bound and Reference-Bound

The **Class** aspect applies to both objects and references, but differently. An object has a certain class, which is fixed upon creation of the object and throughout its lifetime. A reference also has a class, defining which class of object you can assign to the reference. The class of an *object* can never change. The class of a reference can be changed, but only while the reference is **Nothing**. You can not change the class of a reference when an object is assigned to it, except that you can set it to **Nothing** or to the same class as the object again.

Even though the **Object-Class** and **Reference-Class** are the same or one or the other is not filled in, objects and references still each need to separately store which class they have.

##### Pointer-to-Pointer Situations

In a standard situation the **Use As Class, Class Set**, **Reference-Class Get** and **Object‑Class Get** commands are about making an object function as another object’s class. However, you can also make something’s class be yet again another reference. That means that another parent object determines the eventual class.

(However, this might create difficulty for the system to maintain a constant class. You might want another parent to determine the initial class, but the class of an object should not change during its lifetime.)

##### Set Class to Reference

To be able to set the **Class** aspect to another related item, **Class Set** has two overloads:

**Class Set** 🡪 **Set Class to Other Related Item**

**Class Set** 🡪 **Set Class to Other Related List Item**

If you want a single name to express both situations, you could call it **Set Class to Reference**.

##### Get Class which is a Reference

Because the **Class** aspect can be set to another related item, the **Class Get** command gets extra overloads. Next to that, there are different overloads for the two types of **Class Get**: **Reference-Class Get** and **Object-Class Get**. This creates the following overloads:

**Reference-Class Get 🡪 Get Reference-Class which is Another Related Item**

**Reference-Class Get 🡪 Get Reference-Class which is Another Related List Item**

**Object-Class Get 🡪 Get Object-Class which is Another Related Item**

**Object-Class Get 🡪 Get Object-Class which is Another Related List Item**

You could also call them **Get Class which is a Reference**.

##### Use Reference As Class

The **Reference** aspect can be access-controlled for different ways you can use it. Pointer-to-pointer situations require you to be able to use a reference as a *class*. To be able to access control the different purposes for which you can use a reference, the **Reference Get** command gets the secondary implementation:

**Use Reference As Class**

which delegates directly to the **Reference Get** command.

##### The Overloads Recapitulated

Do not wreck your brain over all this delegation and overloading. It is just for pointer-to-pointer situations to have the same command names as standard situations, and also to be able to separately access-control the specific *uses* of references or objects. You will not usually see the pointer-related commands, because they will be implicitly delegated to by the main commands. This leaves us with the following commands:

**Use As Class**

**Class Set**

**Reference-Class Get**

**Object-Class Get**

**Use Reference As Class**

Detail: For that last command you might want to overload **Object Get.** But that does not work. You can not overload it, because they will both take a pointer to an object as an argument. To disambiguate, they have to have a different name and you have to point to a *specific* command.

#### The Extra Commands & Overloads

The system commands for the **Reference**, **Object** and **Class** aspects introduce accessory commands and overloads. They seem to be making the explanations more complicated, but they actually make things easier to work with. There are three reasons for the introduction of the extra commands and overloads:

- Access-control usage

- Common commands for related items and related list items

- Common commands for direct pointers and pointers-to-pointers

Here follows an overview of which reason applies to which command or overload:

##### Access-control usage:

**Use Reference As Object**

**Use Reference As Class**

**Use As Class (= Use Object As Class)**

##### Common commands for related items and related list items:

**Reference Get 🡪 Related Item Ge*t***

**Reference Get 🡪 Related List Item Get**

##### Both:

*- Common commands for related items and related list items and*

*- Common commands for direct pointers and pointers-to-pointers*

**Object Set** 🡪 **Set Object to Other Related Item**

**Object Set 🡪 Set Object to Other Related List Item**

**Object Get 🡪 Get Object which is Another Related Item**

**Object Get 🡪 Get Object which is Another Related List Item**

**Class Set 🡪 Set Class to Other Related Item**

**Class Set 🡪 Set Class to Other Related List Item**

**Reference-Class Get 🡪 Get Reference-Class which is Another Related Item**

**Reference-Class Get 🡪 Get Reference-Class which is Another Related List Item**

**Object-Class Get 🡪 Get Object-Class which is Another Related Item**

**Object-Class Get 🡪 Get Object-Class which is Another Related List Item**

Again: the reasons for extra commands, overloads and delegation are:

- Access-control usage

- Common commands for related items and related list items

- Common commands for direct pointers and pointers-to-pointers

#### System Commands for the Value Aspect

The **Value** aspect is controlled through two system commands:

**Value Get**

**Value Set**

**Value Get** gets the value of an object.

**Value Set** sets the value of an object.

#### System Commands for the Clone Aspect

The **Clone** aspect is controlled through two system commands:

**Clone Get**

**Clone Set**

The **Clone** aspect is relate to the **Value** aspect, but will also copy the sub-objects’ values. **Clone** has a parameter, that defines the cloning depth: the depth at which to clone sub-objects. When the cloning depth is **2**, then this is also expressed as:

**Clone (2) Get**

**Clone (2) Set**

**Clone Get** copies the object and sub-objects. **Clone Set** assigns the cloned values to another object. This can also be a new object.

#### System Commands for the Name Aspect

The **Name** aspect is controlled through two system commands:

**Name Get**

**Name Set**

**Name Get** gets the name of an object, reference or list.

**Name Set** sets the name of an object, reference or list.

#### System Commands for the Data Aspect

The **Data** aspect is controlled through two system commands:

**Data Get**

**Data Set**

**Data Get** and **Data Set** can not be called. They can only be access-controlled. By access‑controlling them you are access controlling being able to read or write to any sub-object or deeper object. The **Data Get** command abstractly represents any *read*, in other words: any possible **Get** call to any sub-object. The **Data Set** command abstractly represents any *write* to any sub-object.

#### System Commands for the Execute Aspect

The **Execute** aspect is controlled through one system command:

**Execute**

You can access control being able to execute a command. Some commands are never meant to be executed, because they are a definition.

#### System Commands for the Existence Aspect

The **Existence** aspect is controlled through the following system commands:

**New**

**Annul**

The **New** command has an optional **Class** argument, with which you can indicate the class to create a new **Object** from.

There is also the system attribute **Is Something** which returns **True** or **False**, also represented by the terms **Something** or **Nothing**.

#### System Commands for the List Aspect

The **List** aspect is controlled through the following system commands:

**Add**

**Remove**

The **Add** command is part of a **List** object. The **Add** command adds an item to the list. This may be an existing item, passed through the optional argument **Item**. You can also create a new item for it, possibly of a new class, using the **New** command, which is part of the **Existance** aspect.

The **Remove** command is part of a **Related List Item**. It removes that item from the list.

### System Interfaces

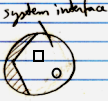
#### Main Concept

##### System Interface Notation

Usually you see an object’s members:



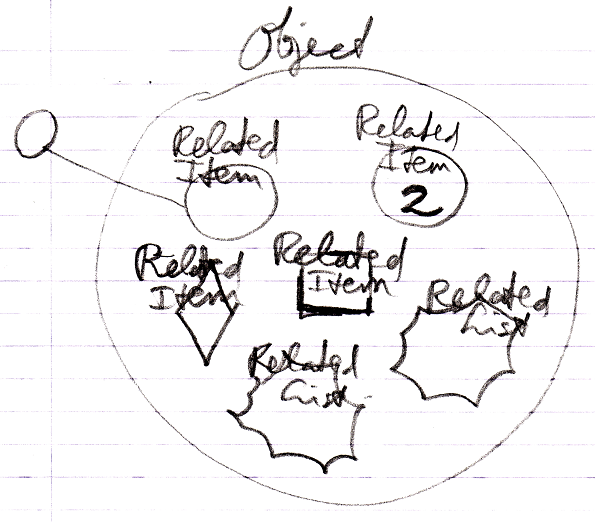
But all objects, references and lists are based on system objects. You can also show the members of the system object instead. You can do that by breaking open the inner workings of the objects and show the *system interface*:



When the system interface is shown, the normal members are not visible anymore.

##### Example: System Interface of an Object

An **Object** might normally look like this:



But if you open up an **Object's** system interface, then the related items and related lists are shown as the **Related Items & Related Lists** collections:



The **Related Items** and **Related Lists** are now displayed circles: normal objects, even when they are *commands* and even when they are *lists*. This only represents the exact way an **Object** actually internally works.

##### Aspects

Next to controlling *sub-objects*, the system interface also controls *aspects*, for instance, the **Value** aspect:



##### System Interface = Publics of System Objects

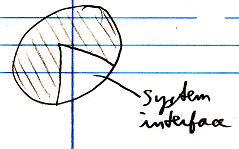
Everything about system interfaces is based on the fact that the system interface is nothing more than the public members of a system object. Even when the explanation about system interfaces is incomplete, what is missing can be derived from the fact, that the system interface will simply show all the public members of a system object. You should also realize that the *private* workings of system objects are not shown in the system interface.

##### Origin of the System Interface Notation

The notation of the *system interface* is derived from the *interface* notation. When an object has several interfaces, each interface is shown as a triangle inside the object:

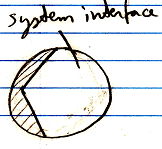


But showing the system interface is like showing the interior ‘of the symbol itself’. Therefore, the triangle of the system interface is now stuck to the border of the symbol:



That way the interface is more part of the object itself, rather than being a sub-object.

To make more room inside the system interface the following, exaggerated notation is used:



It also looks as if you have broken open the shell of the symbol to see its inner workings. It looks like you have opened up the system and can see its internal wiring. The system interface is like seeing the setup of the symbol machine.

#### System Interface of an Object

The main thing you see when showing the system interface of an **Object** is the **Related Items & Related Lists** collections.



But apart from sub-objects, an object has the following aspects:

**Value**

**Class**

**Execute**

**Clone**

**Data**

They are controlled through system commands. Those commands will also be visible inside the system interface.

##### The Value Aspect in the System Interface

The **Value** aspect is controlled through two commands:

**Value Get**

**Value Set**

The **Value** aspect is represented by a triangle, that wraps together the members to control the **Value** aspect:



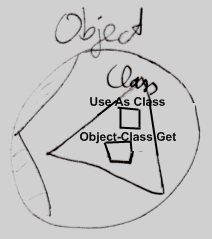
##### The Object-Class Aspect in the System Interface

The **Class** aspect has *five* system commands, but only *two* of them apply to **Objects**. The other ones apply to references. The **Class** aspect of an **Object** is controlled through the following commands:

**Use As Class**

**Object-Class Get**

The commands are placed inside a triangle, that wraps together the members of the **Class** aspect:

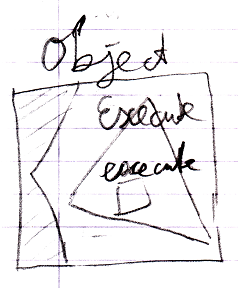


##### The Execute Aspect in the System Interface

The **Execute** aspect only applies to executable objects, also called commands. The **Execute** aspect only has one command:

**Execute**

The command is placed inside a triangle, that wraps together the members of the **Execute** aspect:



##### The Clone Aspect in the System Interface

The **Clone** aspect is controlled through two commands:

**Clone Get**

**Clone Set**

Both commands have a **Depth** parameter to indicate the cloning depth.

The commands are placed inside a triangle, that wraps together the members of the **Clone** aspect:



##### The Data Aspect in the System Interface

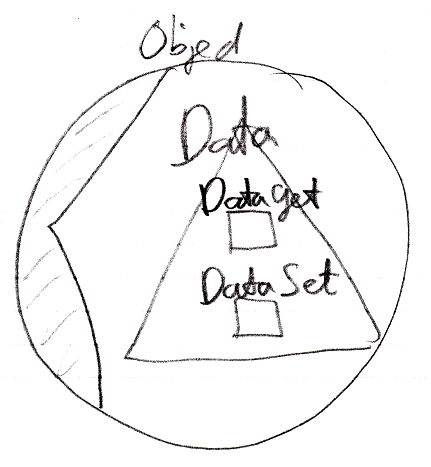
The **Data** aspect is controlled through two system commands:

**Data Get**

**Data Set**

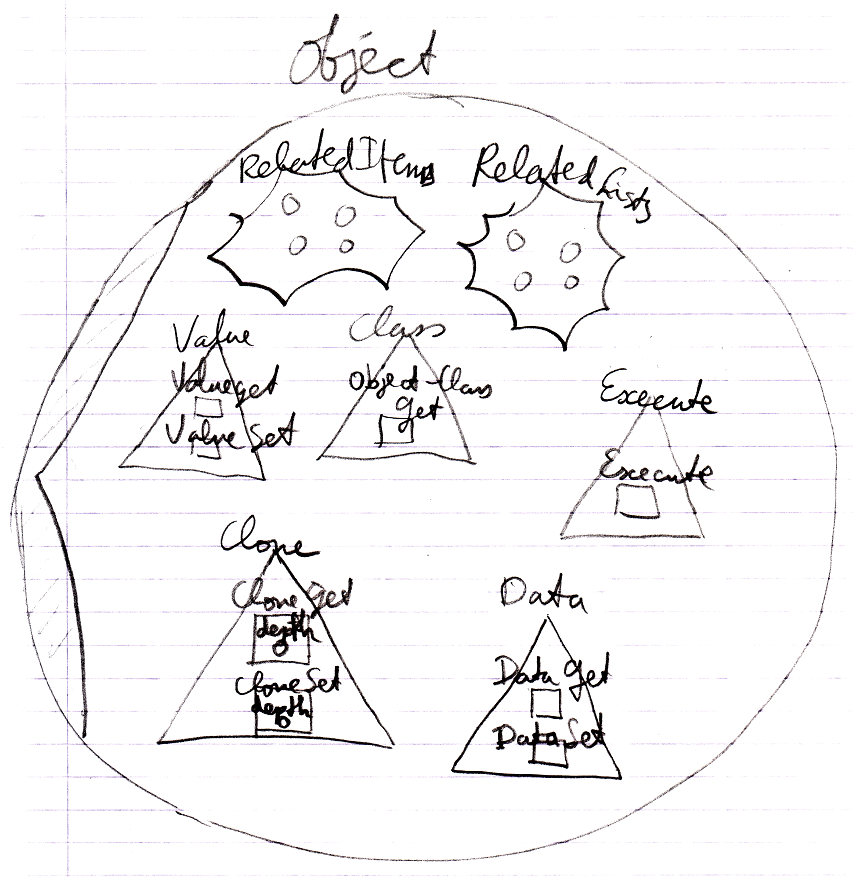
The **Data Get** and **Data Set** command can not be called; they can only be access-controlled to control read-write access to the object and all the contents of the object.

The commands are placed inside a triangle, that wraps together the members of the **Data** aspect:



##### The Full System Interface for Object

The full system interface of an **Object** looks like this:



#### System Interface of a Related Item

When you open up the system interface for a **Related Item**, you get to see the system commands that apply to **Related Items**. Aspects, that apply to a **Related Item** are:

**Reference**

**Object**

**Existance**

**Class**

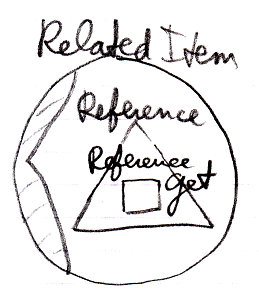
**Name**

##### The Reference Aspect in the System Interface

The **Reference** aspect is controlled through only one command:

**Reference Get**

The **Reference** aspect is placed inside a triangle, that wraps together the members to control the **Reference** aspect:



##### The Object Aspect in the System Interface

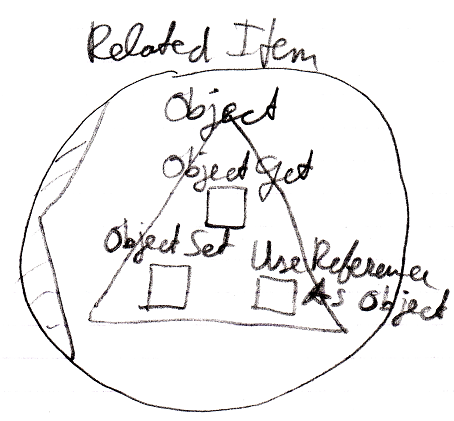
The **Object** aspect of a **Related Item** is controlled through several commands:

**Object Get**

**Object Set**

**Use Reference As Object**

The commands are placed inside a triangle, that wraps together the members of the **Object** aspect:



##### The Existence Aspect in the System Interface

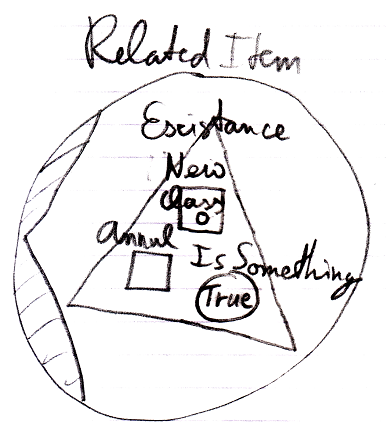
The **Existence** aspect is controlled through two commands and an attribute:

**New**

**Annul**

**Is Something**

The members are placed inside a triangle, that wraps together the members of the **Existence** aspect:



##### The Reference-Class Aspect in the System Interface

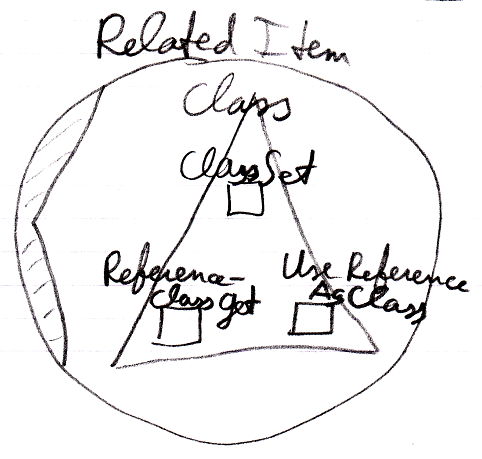
The **Class** aspect has *five* system commands, but only *three* of them apply to **Related Items**:

**Class Set**

**Reference-Class Get**

**Use Reference As Class**

The commands are placed inside a triangle, that wraps together the members of the **Class** aspect:



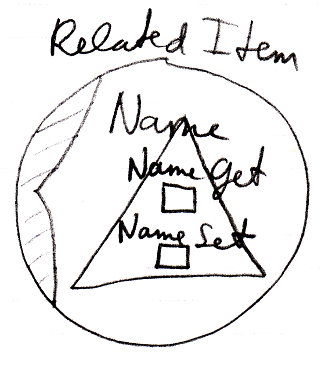
##### The Name Aspect in the System Interface

The **Name** aspect is controlled through two commands:

**Name Get**

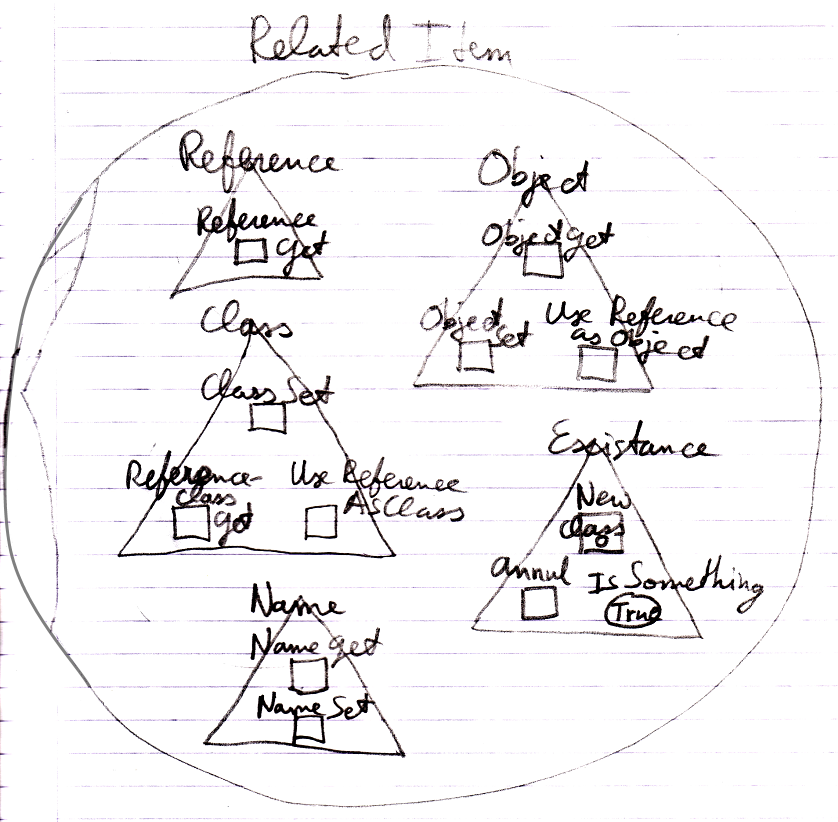
**Name Set**

The commands are placed inside a triangle, that wraps together the members of the **Name** aspect:



##### The Full System Interface for Related Item

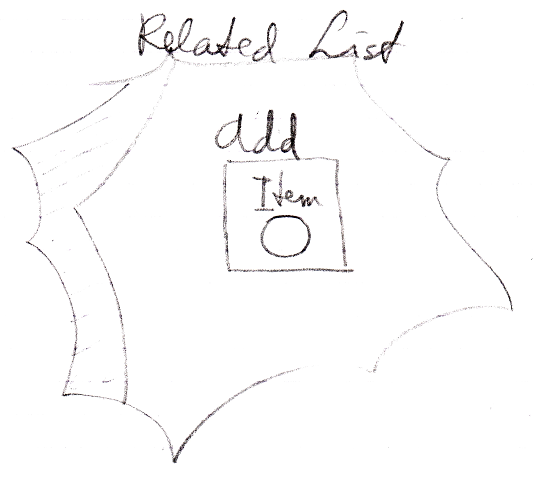
The full system interface of a **Related Item** looks like this:



#### System Interface of a Related List

When you open up the system interface for a **Related List**, you get to see the system commands that apply to **Related Lists**. Only the **List** aspect applies to **Related Lists**. The system interface of a **Related List** will show the **Add** command. The **Add** command has an optional **Item** argument, to add an existing item to the list.

So the system interface of a **Related List** looks like this:



But more members may be introduced later. Specifically you would probably would want to centrally control the **Class** aspect of all the items in the list.

#### System Interface of a Related List Item

When you open up the system interface for a **Related List Item**, you get to see the system commands that apply to **Related List Items**. Aspects, that apply to a **Related List Item** are:

**Reference**

**Object**

**Existence**

**Class**

**List**

Those are the same as a **Related Item**, minus the **Name** aspect, plus the **List** aspect.

Only the **List** aspect for the **Related List Item** is explained here. The other aspects were already worked out in the article *System Interface for Related Item*.

##### Related List Item’s List Aspect in the System Interface

The **List** aspect of the **Related List Item** is represented by:

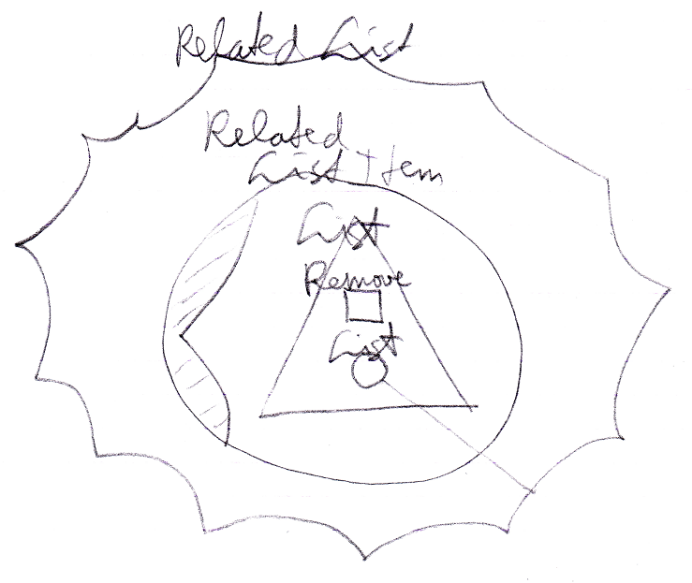
**Remove**

The command, which removes the item from the list

**List**

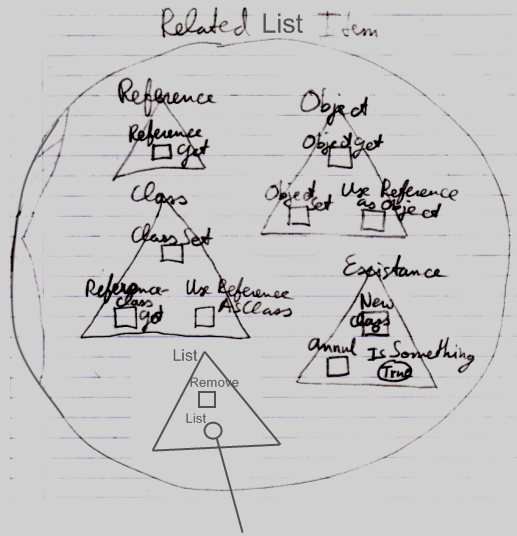
A reference to the **List** that the **Related Item** it is part of.

The members are placed inside an interface, that wraps together the members of the **List** aspect:



##### The Full System Interface for Related Item

A **Related List Item** is almost the same as a **Related Item**, so the full system interface of a **Related List Item** also shows all the other aspects:



#### Use-Command Gets Another Aspect

The **Use As Class** command is part of the **Class** aspect but **Gets** the **Object** aspect.



A **Use** commands **Gets** a different aspect than what it applies to. The **Use Reference As Object** command is part of the **Object** aspect but **Gets** the **Reference** aspect. The **Use Reference As Class** command is part of the **Class** aspect, but **Gets** the **Reference** aspect.

How a **Use** command delegates to a **Get** command of the other aspect is not visible in the system interface, because that is private implementation. You only get to see the public members of the system objects, not their implementation.

#### Aspect-In-A-Triangle

An aspect is represented by a triangle containing system commands:



It was a design choice to have system commands placed inside a triangle, that represents the aspect, because it does not always look best to explicitly qualify each **Class** command with ‘**Class .**’ So you can directly call the **Use As Class** command and not use something like:

**Class . Use As Class**

**Class . Use Object As**

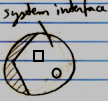
**Object . Use As Class**

That last one *looks* fine, but places a class-related action inside the **Object** aspect, which you do not want, because you want to keep all commands that apply to the **Class** aspect together under the **Class** aspect.

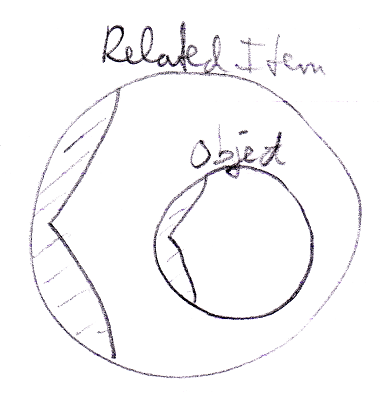
All-in-all: this is a design choice.

#### System Interfaces of Objects and References

There are system commands that apply to **Objects** and system commands that apply to **References**, but when you show the system interface of a symbol, which do you show? The system interface of the **Object** or the system interface of the **Reference**?



The solution for this problem is that you always deal with *references* to objects, never with the object directly, so when you open up the system interface of a symbol, the system interface of the **Reference** is shown. However, inside of it you will find the referenced **Object**, showing its system interface.



#### Preliminariness of the System Interface Notation

The basics of the system interface notation are not preliminary. But the system interface shows the exact public members of the code base objects. If the code base were to be programmed differently, the system interface members will also look different. Design choices were made in this documentation, which influenced the notation of the system interface members. Perhaps when implementing the code base in the future, different design choices are made and that will change the way the system interfaces look.