|  |
| --- |
| Circle Language Spec: Inheritance |

## Specialization

There are four groups of specialization techniques that can be identified. Each will be separately explained:

*Data Replacement*

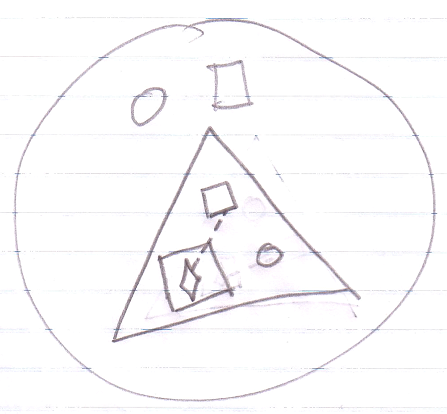
*Altering the Member Set*

*Detouring Members*

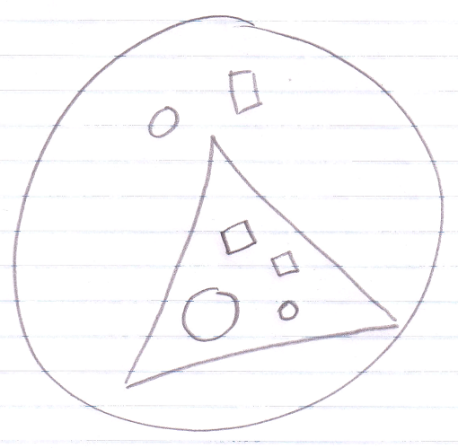
*Altering Command Implementations*

### Specialization & Data Replacement

You do not need any specialization techniques to alter the behavior of a base object.



Anything inside the base object is just data, that can be changed, including the commands, command calls and member sets. This is all just data that you can change.



One condition though is that the data is accessible.

However, data replacement is not what we usually call specialization. What we call specialization are less destructive methods where most of the data stays in tact.

Some specialization techniques actually replace data, rather than just add it, but they are generally considered less destructive than sheer data replacement. *Data replacement* and *specialization* are two different concepts.

### Altering the Member Set

The easiest group of specialization techniques is *altering the members set*. Three techniques can be identified:

*Member Addition*

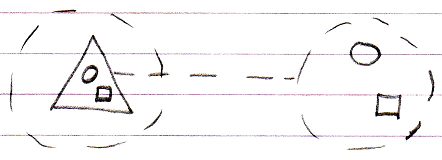
*Member Exclusion*

*Member Inclusion*

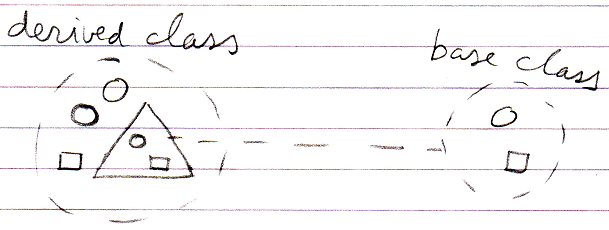
Each technique is explained in a separate article

#### Member Addition

The easiest technique for specialization by altering the member set is *member addition*. Here the concept of class inheritance is displayed:

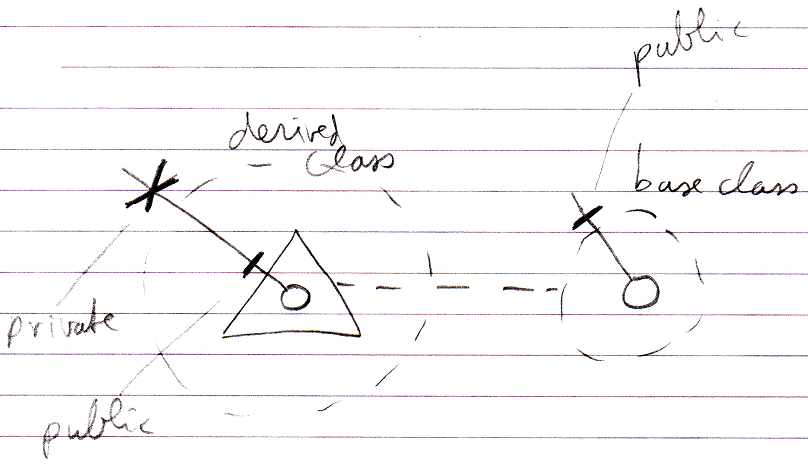


Member addition means adding a member, that was not there in the base, but will be in the derived object:

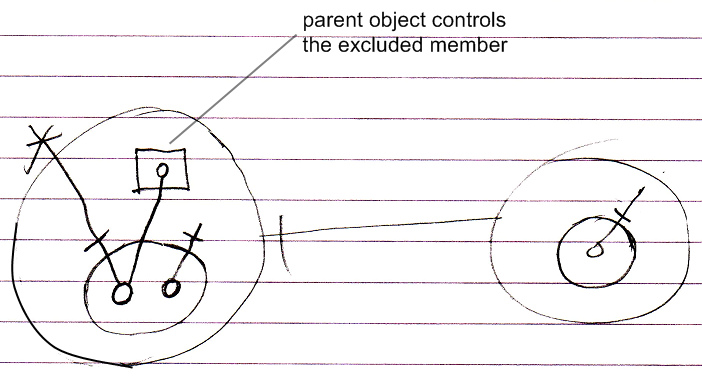


#### Member Exclusion

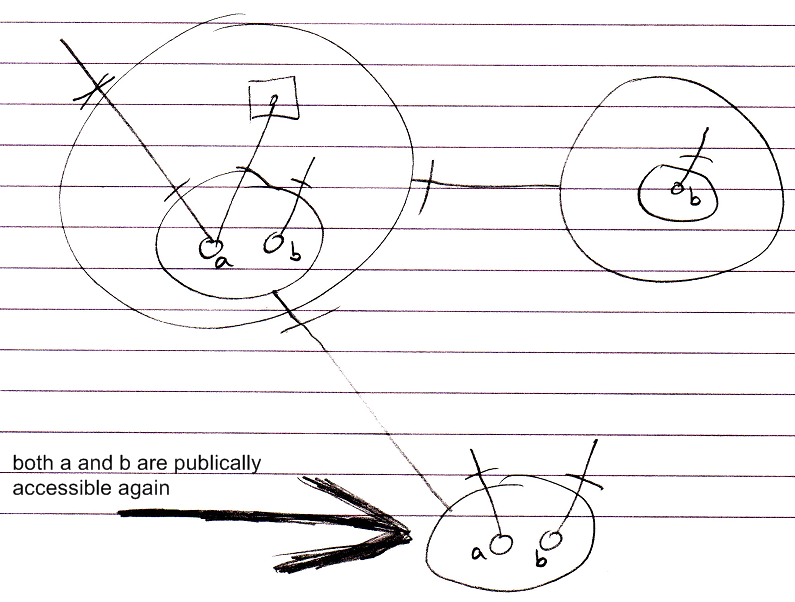
Another technique for specialization by altering the member set is *member exclusion*. This is not possible in most other object oriented languages. What happens is that members from deeper objects that were formerly Public, are Private through the derived class.



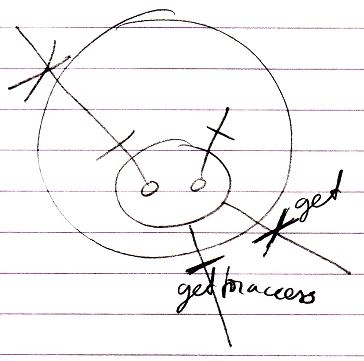
This also makes it possible to protect some of the members of the sub-objects, and let the parent object be the only one to control them.



Do note that this protection is dependent on the context, so when you can reference the object from another location, you can still access the excluded member:



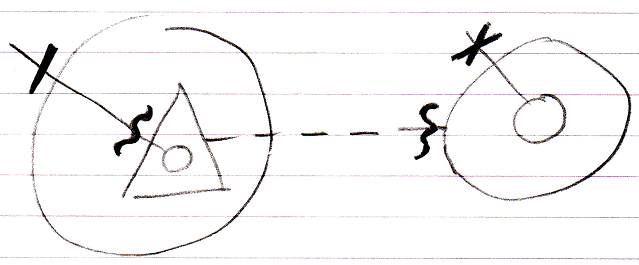
You can prevent this by making the Object Get Private while Object Get For Access is still Public.



(there is no notation yet for the distinction between Object Get, Object Get For Access and Object Get For Copy)

#### Member Inclusion

Another technique for specialization by altering the member set is *member inclusion*. This is the opposite of member exclusion. Instead of making a formerly Public member Private, you make a formerly Private member Public. This seems to go against the rules for encapsulation, but in fact it is valid, since only Friend objects can do it.



### Detouring Members

Detouring members is a group of specialization techniques. These techniques allows a derived class to redefine a member from the base class. This can happen in two ways:

*Shadowing*

*Overriding*

Each of them is covered in a separate article

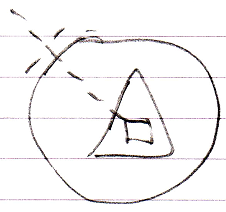
#### Disclaimer

Inheritance makes extensive use of the concepts of overriding and shadowing. However, the notations and implementations of overriding and shadowing are not worked out to perfection yet. Preliminary notations are proposed. They will be reconsidered when working out the chapter *Object Resolution*.

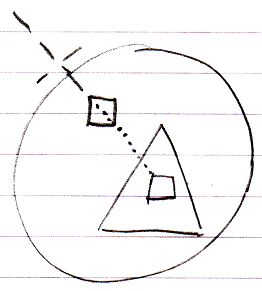
#### Shadowing

A derived class can *shadow* a command of the base class. This means that when you call a base command through the derived object, the derived command is run instead.

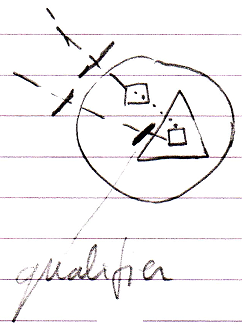
Below you see a normal base member and a call to it:



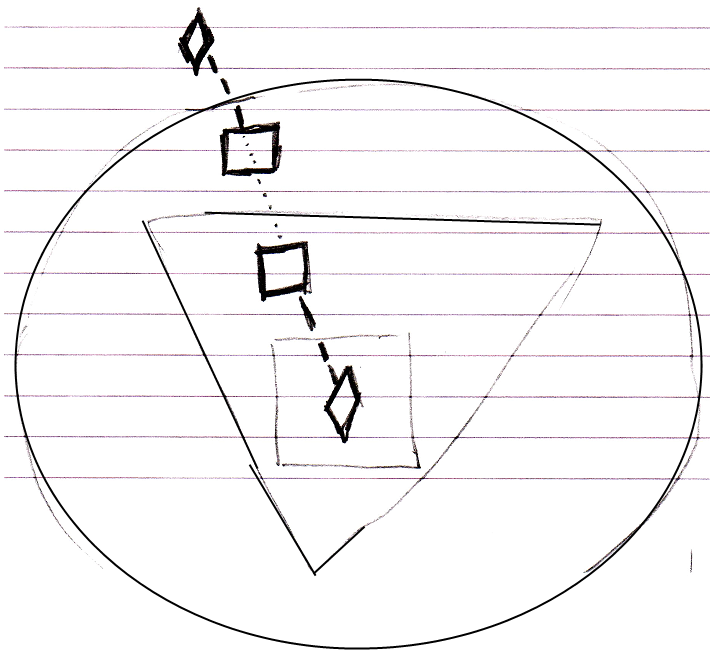
Below you see the situation when the base command is shadowed:



You can still call the base command, by qualifying the call so that the interface is adressed directly:



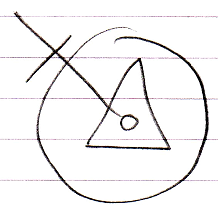
When the base class itself calls the shadowed command, the original base command is called.



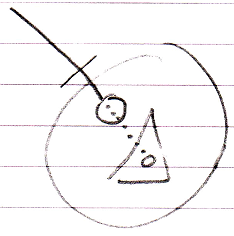
This is where the behavior of *shadowing* is different from *overriding*.

What should be mentioned is that you can also shadow objects, just like you can shadow commands.

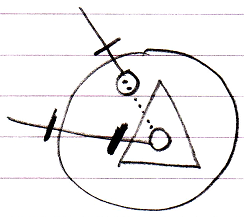
*Unshadowed:*



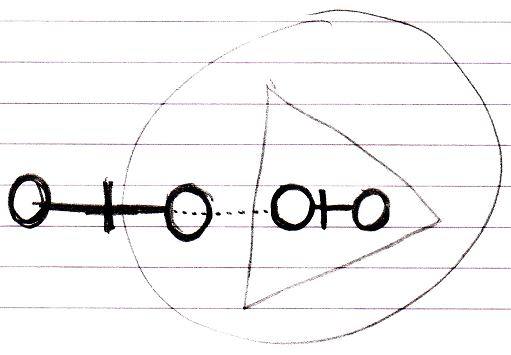
*Shadowed:*



*Access to base member through qualifier:*

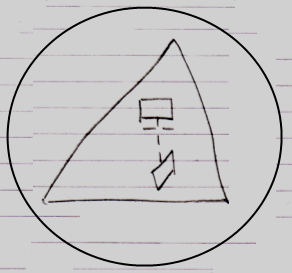


*Base class calls base member, not derived member:*

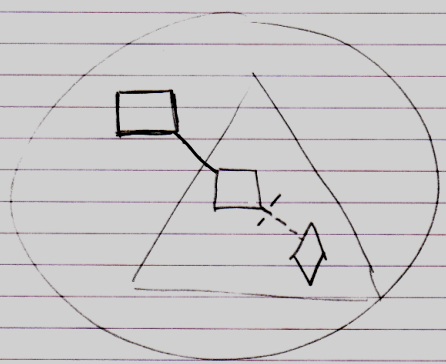


#### Overriding

Base classes normally call their own commands:



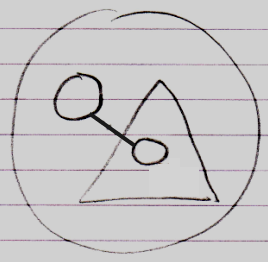
But derived classes can redefine those commands. When the base class tries to call its own command, the call is detoured to the redefined command. In other words: the original call to the command is detoured to another command.



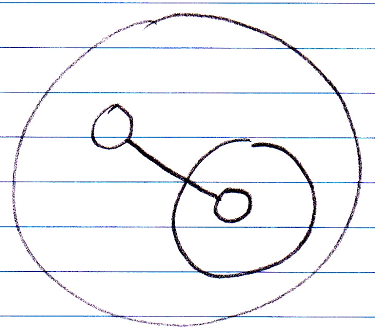
The base class can not run its original command anymore, but the command is detoured to a command defined in the derived class.

Overriding is accomplished with simple object redirection.

What should be mentioned is that you can also override objects, just like you can override commands:

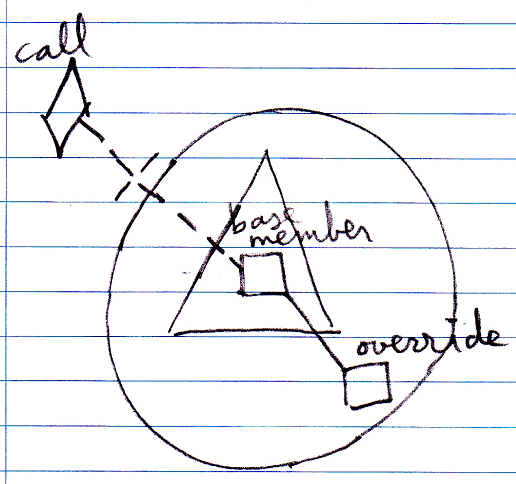


What should also be mentioned is that overriding is not specific to inheritance. It can also be done in case of regular containment:

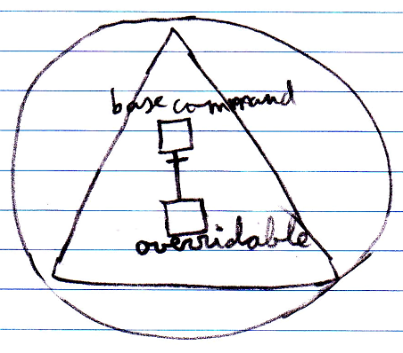


##### Calling Base Class’s Version of Overridden Member

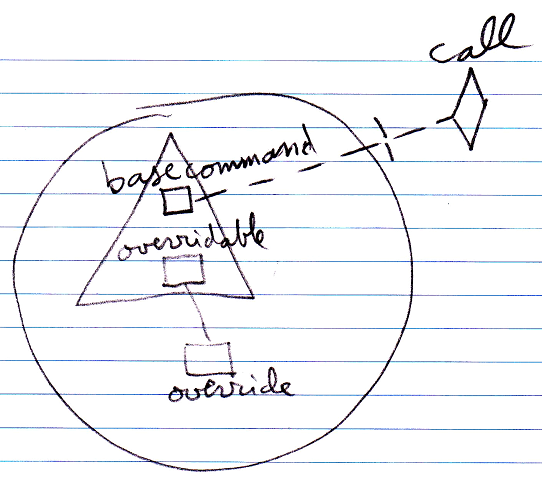
Some programming languages offer a way to call the member defined by the base class, even when the base member was overridden. In the new computer language this is not possible by default, because any call to the overridden member will redirect to the overriding member.



However, there is an alternative: define a base command and create an overridable command that is a reference to the base command.

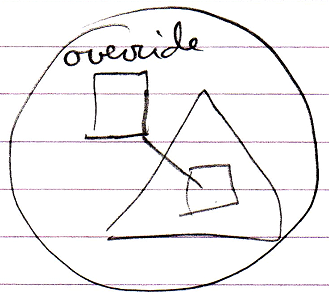


You can replace the overridable command, but you can still call the base command.



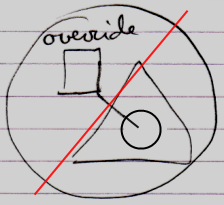
##### Overriding Alternative: Event Notation

An alternative notation to overriding is the override event.



An override event is basically unnecessary, because you can already accomplish that by simple object redirection, but the notation is intuitive, and creates an analogy with the notation for pre- and post-extension, which is introduced later. The notation also offers the possibility to easily express overriding *system* commands, also explained later.

The notation does not seem to work for object overriding.

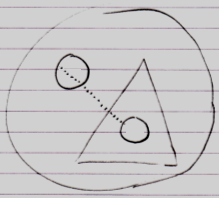


This is because object overriding redirects to an object, and since events are commands, it looks strange.

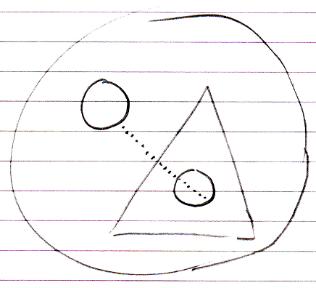
##### Discarded Intrinsic Overriding Notation

There used to be an intrinsic overriding notation which looked a lot like shadowing:

*Shadowing notation:*



*Intrinsic overriding notation:*



However, the intrinsic overriding notation was discarded, because it is undefendable to have it when the same is possible with simple object redirection.

### Altering Command Implementations

Another way to specialize a derived object is to *alter* the command implementations of the base object (as opposed to replacing a command entirely).

Several separate techniques can be identified and will be explained in individual articles:

*Command Extension*

*System Command Extension*

*System Command Overriding*

*System Command Shadowing*

Furthermore there are a some of variations on these techniques which also have a separate article:

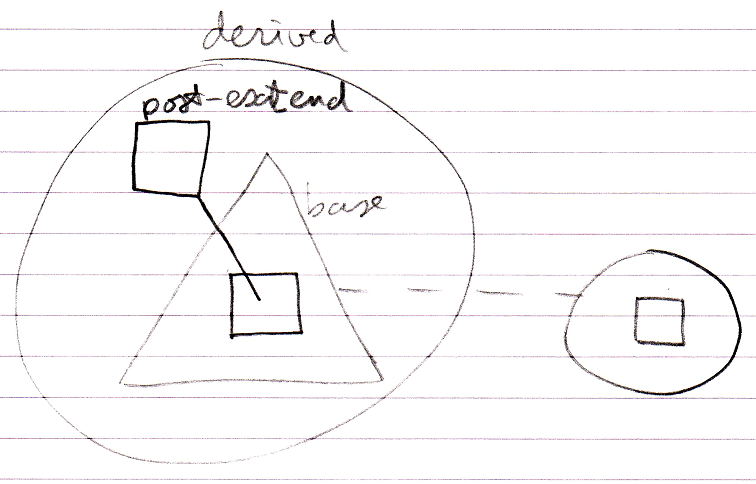
*System Command Extension By Shadowing*

*System Command Extension By Overriding (Questionnable)*

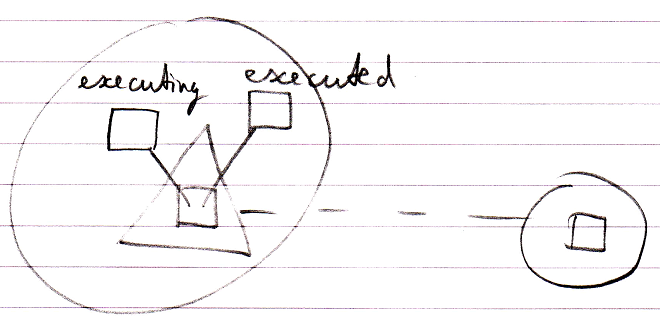
*System Command Overriding By Extension*

#### Command Extension

When a base class defines a command, the derived class can add extra functionality to the existing command. The derived class can pre- or post-extend a command from the base class:

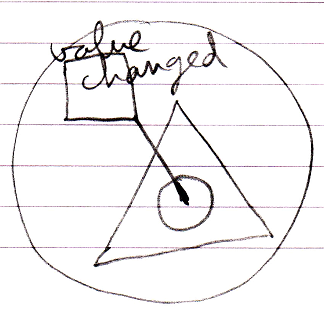


This pre- and post-extension happens by means of the already available system events explained in the *Events* chapter. The events are actually named Executing and Executed.



#### System Command Extension

Another way to specialize a derived object is to let it define extra functionality around a *system* command. The previous section was actually a demonstration of how to extend the implementation of the Execute system command. But *other* system commands can be extended as well, such as the Value Set command. Here is an example where the Value Set command is post-extended by implementing the Value Changed event.



So then the derived class defines something that will be run every time the object’s value changes.

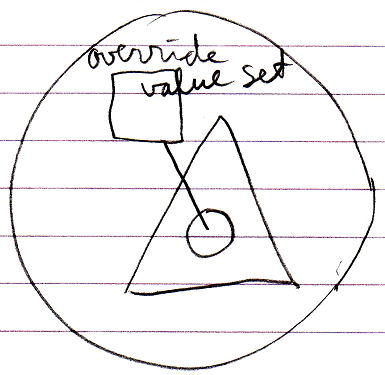
#### System Command Overriding

Overriding was already introduced before, but what is separately mentioned here is that you can override any of the system commands. You can for instance override the Value Set command.

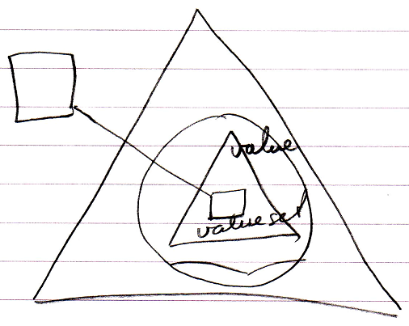
A lot of notations were considered but many were complicated or ambiguous. Easier alternatives are adopted here.

Here are the two alternatives to a notation for overriding system commands:

*The event notation*

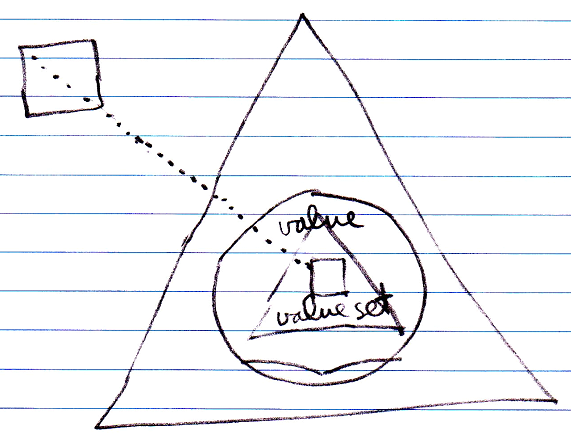


*System interface with object redirection*



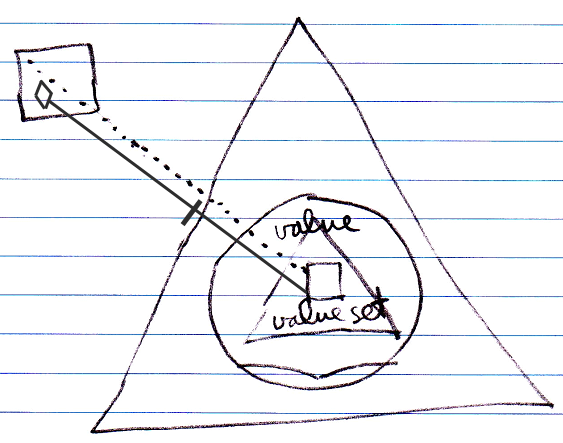
#### System Command Shadowing

Shadowing a system command can only be displayed by opening up the system interface:



#### System Command Extension By Shadowing

You can accomplish command extension by employing the shadowing concept. The idea is that you shadow the base command, executing your own procedure and call the base command from your own procedure.



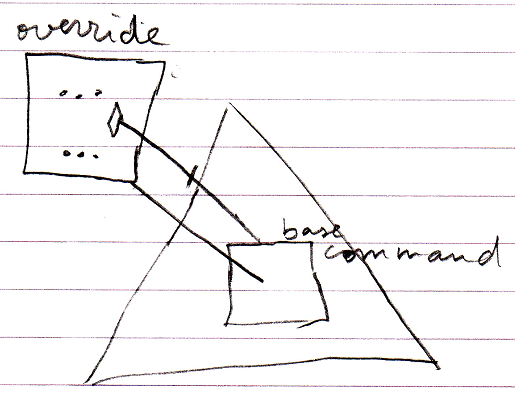
Note that the call to the base command has to be qualified with the base object.

When anyone tries to call Value Set, the shadowing command is run instead, unless someone can qualify the call with the base object.

A benefit of system command extension by shadowing compared to normal system command extension is that these extensions only count inside the derived class.

#### System Commands Extension By Overriding (Questionable)

This is a technique that will not work. The idea is that you accomplish command extension by employing the overriding concept. The idea is that you override the base command, executing your own procedure and call the base command from your own procedure.

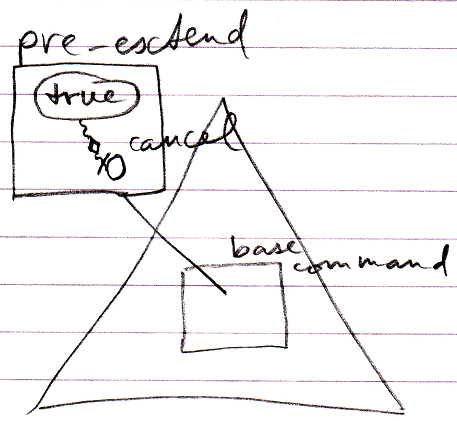


The problem with this technique however, is that the call to the base command immediately detours to the derived command again, causing an infinite loop.

However, extending system commands by *shadowing* does work.

#### System Command Overriding By Extension

There is an alternative to overriding: you can implement the pre-extension event of a command and cancel the event.



This cancels the execution of the base command.

### Destructive & Non-Destructive Specialization Methods

Most of the specialization methods are extensions. The extensions are commonly *data* that is *added* to the base system. There are two specialization techniques that do not only extend, but also *change* the behavior of the base:

- Overriding

- Pre-extension with cancellation

These could be considered destructive specialization methods, because they change behavior, while *extensions* are mere non-destructive specialization methods.