#### Symbol: Generiek sluit daarbij aan

Het type van de items is nu wel heel generiek. Dat sluit aan bij generiekheid in Symbol. Mooi.

#### Ideas

Computer Language,

Object symbols can be prototyped as classes and interfaces.

You can also access control object symbols, so that they can only be used as a class or an interface.

JJ

Diagrams,

I think it is a nice notation to draw a circle with dashed or dotted line, when it can only be used as a class or only used as an interface or if it IS used as a class, and IS used as an interface.

In that case, you can display several different lines for the symbol

JJ

Classes & interfaces implementation

Classes, interfaces en ook relations kunnen concepts worden,

in plaats van principles, omdat het slechts om reenforcement gaat:

je pint een stuk structuur van het object vast aan een ander object.

De code base kan ook zonder.

Dat geld dan voor veel van wat nu Coding Principles zijn.

If classes is to be implemented as a relation between objects, then you already need relations between object, even though you don’t have relation between classes yet.

JJ

> An object's class or a class's interface is a prototype-based principle,

so any other object can serve as a class or interface,

but symbols will be more likely to be assigned a fixed role.

> In Coding Principles, Classes and Coding Principles, Interfaces:

In the coding principles documentation,

for class you need to clarify that object with the

same class have the same public and private symbols,

but the state of each object is different, meaning

that each object has different attribute values and subobjects point

at different targets.

A similar explanation has to be given for the interface coding principle.

Classes & Interfaces,

You should introduce type safety and genericity sooner,

in the coding principles.

Type safety can be implemented as a concept,

but assigning a class or interface really doesn't make

any sense without it, so you have to introduce it

in the articles about classes and interfaces.

A classes fixes the collection of sub objects that

exist in it, and the names for those sub-objects.

A sub-object can reference another object,

but its names is fixed by the class.

Usually the sub objects defined by the class

also have a fixed class.

Explain everything in terms of classes first,

not mentioning interfaces.

JJ

Classes & Interfaces,

You have to explain sooner, what's the use of fixing

the class of an object reference.

You have to discuss what's the use of fixing all classes,

and what is the use of not fixing any classes.

Fixing the classes of objects and their contents

results in exactly knowing how the an object looks and behaves.

Fixing the interface of objects results in exactly knowing

how an object looks, even though the behavior can be different.

If you pass an object along to a procedure, the procedure can

make no assumptions about the contents of the object,

unless it has a fixed class or interface.

Type safety only useful so PROCEDURES can make

assumptions about the interface or behavior of an object.

But when just using an object as a folder, you keep the way

the object looks totally variable, so you can put anything in it.

You won't be running a procedure over it anyway.

JJ

## from original Symbol documentation

### Object Basics

#### Same Object, Same Type

##### Same Object, Same Type

If you make two symbols the same object:



They automatically are the same type, sort of like this:



##### Same Type, Same Interfaces

When you make two symbols the same type:



They (by default) automatically get the same interfaces, sort of like this:



##### Same Object, Same Type, Same Interfaces

If you make two symbols the same object:



They automaticaly have the same type with (by default) the same interfaces, sort of like this:



##### Not Shorthand

However, you can’t call it *shorthand*. The following two are not precisely the same thing:

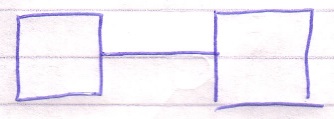




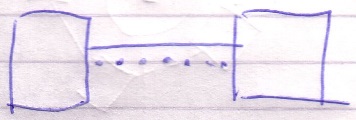
The lack or presence of a type line *does* make a difference, as will be explained in a later chapter.

#### Same Procedure, Same Interface

When you make two symbols the same procedure:



They also get the same interface, sort of like this:



However, the lack or presence of an interface line *does* make a difference, as will be explained later.

### Genericity

Genericity is about the how loose or rigid a system aspect is.

For instance, if a circle has no type line, the object can be of any type. The type is *generic* in this case. If the type line is set, though, the object can only be of a single type. The type is *explicit* in that case.

There are more such phenomena. This section will cover all of them.

#### Object Symbol Genericity

This section covers the all forms of genericity for object symbols.

##### Original and Delegated Object

The first concept I will explain is quite simple.



If a symbol has an object line then it adopts the object of the line target. **A** delegates to **B**. You can also say that **A**’s object aspect is *delegated* or that **A** has a *delegated object*.



If a symbol doesn’t have an object line, though, it represents its own object. The symbol is an *original object*.

Synomyms for the terms *delegated* and *original* are allowed. *Delegated* has synonyms such as *diverted*, *redirected*, etcetera. All are valid. Synonyms for *original object*, can be expressions such as ‘the symbol *is its own object*’.

|  |  |
| --- | --- |
| Original object | Object determined by no line  (no object line) |
| Delegated object | Object determined by a line  (object line) |

The redirection of the object aspect is the main type of redirection. Therefore you can also speak of *the* delegation.

##### Original and Delegated Type

It works almost the same for the type aspect. However, the type aspect can be delegated by both a type line as well as an object line. If a symbol has a type line or object line, then it is said to have a *delegated type*.

|  |  |
| --- | --- |
| Type delegated by object line | Type delegated by type line |

If the symbol has type nor object line, it represents its own type. The symbol is said to have its *original type*.



The type determines the symbolic contents of a symbol. If a symbol has a delegated type, it has the same symbolic contents as its target type. If it has its own original type, then it determines the contents itself.

During execution of a system a symbol can change from having a delegated type to original type and back.

|  |  |
| --- | --- |
| Delegated type | Line removed, original type reappears |

When you remove a type line, the symbol’s original contents reappear. So the original type definition of a symbol isn’t lost as soon as you delegate the type line. You just temporarily can’t see the original type contents. The original type systematics reappear when the type is no longer delegated. When a type is delegated, usually nothing is defined as the original type. So not every object symbol needs to store its own original systematics.

|  |  |
| --- | --- |
| Original type | Type determined by no line  (type nor object line) |
| Delegated type | Type determined by a line  (type or object line) |

##### Explicit and Generic Type

This is the genericity already explained in the introduction to genericity. When a type is undetermined by a type line, the symbol adopts the type of its object target, whatever type this object may be of.

|  |  |  |
| --- | --- | --- |
|  |  |  |

The type is *generic* in this case. If the type line is set, however, the object can only be of a single type.

|  |  |
| --- | --- |
|  |  |
|  |  |

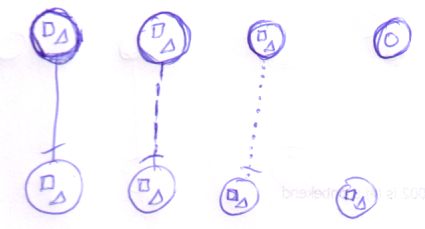
The type is *explicit* in that case. You can also say that the type is *set*.

|  |  |
| --- | --- |
| Explicit type | Type determined by type line  (limits the possible object to objects of that type) |
| Generic type | Type determined by object line  (symbol adopts any type the object might have) |

If the type is determined by no line at all then it has an *original type*. You can speak of neither explicit nor generic in that case. When the type is generic or explicit, it’s always delegated. If the type is explicit, the type is delegated by the type line. If the type is generic, the type is delegated by the object line.

##### Original and Delegated Interface

The interface can be determined by the target of the interface line, type line or object line, but the interface line is most determinant. If any of those line are set, the interface is diverted to the interface target, object target or type target. It is said to be *delegated*. If an object symbol has no interface line, it has its *original interface*.



|  |  |
| --- | --- |
| Original interface | Interface determined by no line  (interface, type nor object line) |
| Delegated interface | Interface determined by a line  (interface, type or object line) |

During execution the interface can change from being delegated to original and back. The original interface isn’t lost as soon as you delegate the interface. When you annull the delegation, then the original interface reappears. It was only temporarily hidden. Mostsymbols don’t define an original interface, so not every object symbol holds original interface systematics.

The use of delegation of the interface is to fix the interface and to use an already defined interface.

##### Explicit and Generic Interface

As the type can be explicitly set, the interface can also be explicitly set. It’s only about the interface set by an interface line; the other line types don’t explicitly set the interface. If the interface is set, then the type or object can only be a symbol whose type at least supports the interface.

|  |  |
| --- | --- |
|  | A |
| B | C |

The top object can point to the left object (picture A) and to the right object (picture B). The left object has the full interface, while the right object supports the interface. Picture C is illegal, because the object target doesn’t support the interface.

If a symbol has no interface line, the interface is said to be *generic*. It adopts the full interface of any type the interface might have, due to type delegation by either a type line or object line.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| Explicit interface | Interface determined by interface line  (one specific interface of the type is used)  (limits the possible types to ones that support the interface) |
| Generic interface | Interface determined by type line or object line  (type’s whole interface used)  (symbol adopts any interface the type might have) |

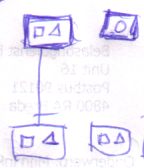
If the interface is determined by no line at all then it has an *original interface*. You speak of neither explicit nor generic in that case. When the interface is generic or explicit, it’s always delegated. If the interface is explicit, the interface is delegated by the interface line. If the interface is generic, the interface is delegated by either type or object line.

#### Procedure Symbol Genericity

This section covers the genericity concepts around procedure symbols.

##### Original and Delegated Definition

When a procedure symbol has a reference line, it adopts the procedure definition of its line target. The symbol has a delegated definition. If the symbol has no reference line, it defines the procedure itself. It has its original definition.



The definition of the procedure symbol determines its symbolic contents. As a system runs, the definition can go from original to delegated and back. When you remove a reference line, the symbol regains its original contents. So the original definition of a symbol is not lost once you delegate the definition. You just temporarily can’t see the original definition. When a definition is delegated, usually nothing is defined as the original definition. So not every procedure symbol needs to store its own original systematics.

|  |  |
| --- | --- |
| Original definition | Definition determined by no line  (no reference line) |
| Delegated definition | Definition determined by a line  (a reference line) |

You can also speak of *original procedure* and *delegated procedure* or *the delegation of a procedure*.

##### Original and Delegated Execution

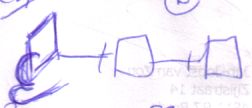
When there’s a diamond somewhere the reference line redirection,



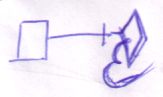
the source symbol of the redirection doesn’t represent its own execution. Its execution is delegated to the other diamond.

|  |  |
| --- | --- |
| Original execution | Delegation doesn’t pass a diamond |
| Delegated execution | Delegation passes a diamond |

Logically, only a diamond can represent its own original execution.



You don’t usually talk about *squares* having an *original execution*, however when a square delegates to a diamond,



the square *is* said to have a *delegated execution*.

##### Original and Delegated Procedure Interface

The interface of a procedure can be delegated to its definition target, but it can also be delegated by an interface line.

|  |  |
| --- | --- |
|  |  |

In both cases the interface is delegated. If a procedure symbol has no line at all, it has an original interface.



During execution the interface can change from being delegated to original and back. The original interface isn’t lost as soon as you delegate the interface. When you annull the delegation, then the original interface reappears. It was only temporarily hidden. Most procedure symbols don’t define an original interface, so most symbols don’t holds original interface systematics.

The *use* of delegation of the interface is to fix the interface of a procedure and to use an already defined procedure interface.

|  |  |
| --- | --- |
| Original interface | Interface determined by no line  (interface nor reference line) |
| Delegated interface | Interface determined by a line  (interface or reference line) |

##### Explicit and Generic Procedure Interface

If a procedure symbol has an interface line then its definition can only be one that has that particular interface. The interface is explicit.

|  |  |  |
| --- | --- | --- |
|  |  |  |

If the interface line of a procedure symbol isn’t set then it adopts the interface of the target definition.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| Explicit interface | Interface determined by interface line  (limits the possible procedures to ones with that interface) |
| Generic interface | Interface determined by reference line  (symbol adopts any interface the procedure might have) |

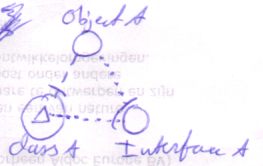
#### Type-Interface Interaction

##### Type Interface Genericity

A special form of genericity is very important and it’s crucial for the use of interfaces at all. I’m talking about: the interface doesn’t determine the type. It’s a genericity always present.

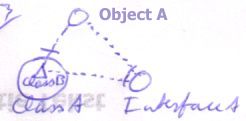
###### Using Interface Extends Type Line

When you set the interface of a symbol then it can only point to definitions that have the interface or support the interface. I will now describe how using a supported interface works.



Here **Object A** has **Type B**. It uses **Interface A** of **Type B**.

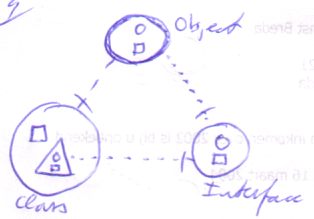
One one hand you can a see using the supported interface as a silent extension of the type line to the triangle with the right interface:



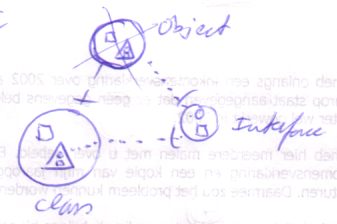
But on the other hand it’s not like that, because that would mean that **Object A** gets **Type B**. It still has **Type A** however. When **Object A** is created, an object of **Type A** is is created, not **Type B**. But **Object A** can only call upon the members of **Interface A**.

###### Object Shows Only Used Interface Implementation

The source object only shows the implementation of the interface it has:

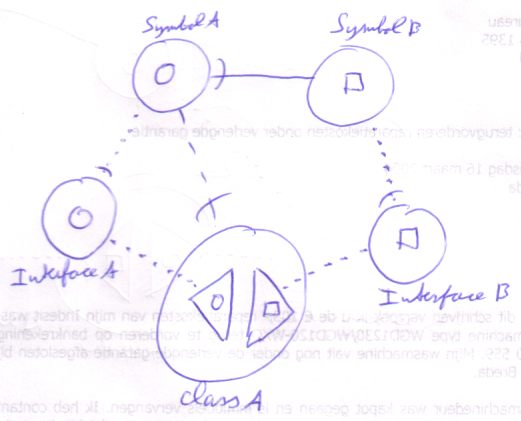


Not the contents of the whole type:



###### Create Symbol: Create Object of the Whole Type

However, when you *create* the symbol, an object of the *full type* is created. This is useful: other symbols may be pointers to the same object, but may use a different interface.



Symbol A is of Type A, but only uses Interface A. Symbol A only shows the systematics of that one interface: a circle. However, it’s an object of the **whole type**. This is important. The importance is pointed out by Symbol B. Symbol B is a pointer to Symbol A, only it uses a different interface. So it shows different systematics.

So object symbols tied together with object lines don’t have to use the same interface at all. The picture above shows that when you use an object line (**Symbol B** points to **Symbol A**), the target doesn’t have to have the same interface, only the same type.

###### Choose From Multiple Types

So one interesting thing about type-interface genericity is that when you’ve set a symbol’s interface line, you can still choose from multiple types.

###### No Interface Line: Follow Object Line

If there is no interface line, the *object* determines *which* interface of the typeis used, as the pictures below demonstrate.

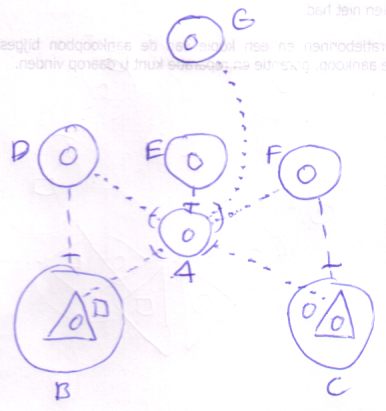
|  |  |
| --- | --- |
|  |  |

**Symbol A** redirects the type to **Type A**. In the first picture **Symbol A** redirects to **Object A**, in the second picture **Symbol A** redirects to **Object B**. **Symbol A**, **Object A** and **Object B** are all of the same type. However, **Object A** uses **Interface A** and **Object B** uses **Interface B**.

This means that the object line determines the interface sooner than the type line. That’s why in the interface trace, the object line is preferred over the type line.

##### Interface Type Generiticy

Another very important use of the separation between types and interfaces is that you can choose any *object* that supports the interface, not worrying about its type.



D, E and F all have different types, while they all support interface A. G also supports interface A. G could point to D, E or F without problems, while D, E and F all have different types. However, G can only call the members of their joint interface. Specialized features of the types D, E and F are unavailable.

Conversely, you can let a program that isn’t familiar with your type, use your type anyway, by supporting an interface that the other program *is* familiar with.

##### Interface Procedure Genericity

Interface procedure genericity is like interface type genericity. It means that if you fix the procedure interface, you can still choose from multiple procedure definitions.

##### Overview of Genericities of Type-Interface Interaction

|  |  |
| --- | --- |
| Interface type genericity | Fixing the interface doesn’t fix the type |
| Type interface genericity | Fixing the type doesn’t fix the interface |
| Interface-procedure genericity | Fixing the procedure interface doesn’t fix the procedure |

#### Final Issues

##### Explicit Isn’t Fixed

When a symbol has an explicit type, it’s object can only be of that type. Having an explicit type, however doesn’t mean that the type of a symbol can’t change. If you change the type line, you’ve changed the type. The only way to make the type immutable at run time is by making it impossible to change the type line at run time. How to do that is explained in a later chapter.

##### Same Object, Same Type

Earlier on, in the section *Object Basics : Same Object, Same Type* I said that if you make two symbols the same object:



they automatically are the same type, sort of like this:

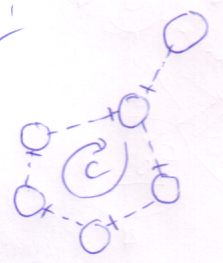


The topics of genericity show that these two pictures aren’t synonyms. In the second picture the type is explicit, while in the first picture the type is generic. The same goes for the other comparisons in that section.

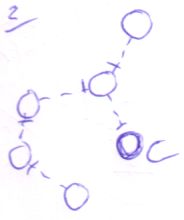
#### Delegated to Original in a Circularity

When you annull a symbol’s type line, it usually regains its original type. All symbols pointing to the reappeared original type, now also have a different type.

In a circularity it’s different. When you delegate the type to a circularity, the whole circularity represents the type.



When you break one of the lines in the circularity, you might think that the one loosing the line regains its original type. The reality is different: no symbols of the former circularity regains its original type.



If you break one of the lines in the circularity, the one loosing the line comes to represent the type and the other members of the former circularity now delegate their type to it.

#### Overview

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Object Symbol Genericity** | | | | | |
|  | **Object** | | | | |
|  | Original object | | Object determined by no line  (no object line) | |
| Delegated object | | Object determined by a line  (object line) | |
| **Type** | | | | |
|  | Original type | | Type determined by no line  (type nor object line) | |
| Delegated type | | Type determined by a line  (type or object line) | |
| Explicit type | | Type determined by type line  (limits the possible object to objects of that type) | |
| Generic type | | Type determined by object line  (symbol adopts any type the object might have) | |
| **Interface** | | | | |
|  | Original interface | | Interface determined by no line  (interface, type nor object line) | |
| Delegated interface | | Interface determined by a line  (interface, type or object line) | |
| Explicit interface | | Interface determined by interface line  (one specific interface of the type is used)  (limits the possible types to ones that support the interface) | |
| Generic interface | | Interface determined by type line or object line  (type’s whole interface used)  (symbol adopts any interface the type might have) | |
| **Procedure Symbol Genericity** | | | | | |
|  | **Definition** | | | | |
|  | Original definition | | | Definition determined by no line  (no reference line) |
| Delegated definition | | | Definition determined by a line  (a reference line) |
| **Execution** | | | | |
|  | Original execution | | | Delegation doesn’t pass a diamond |
| Delegated execution | | | Delegation passes a diamond |
| **Interface** | | | | |
|  | Original interface | | | Interface determined by no line  (interface nor reference line) |
| Delegated interface | | | Interface determined by a line  (interface or reference line) |
| Explicit interface | | | Interface determined by interface line  (limits the possible procedures to ones with that interface) |
| Generic interface | | | Interface determined by reference line  (symbol adopts any interface the procedure might have) |
| **One does not fix the other** | | | | | |
|  | Interface type genericity | | Fixing the interface doesn’t fix the type | | |
|  | Type interface genericity | | Fixing the type doesn’t fix the interface | | |
|  | Interface-procedure genericity | | Fixing the procedure interface doesn’t fix the procedure | | |

## More Ideas

Type Control,

2008-10-20

Strong typing is no more, than a binary format with type checking around it.

It also gives you a nice autocomplete list in your programming environment.

The type checking is only a restriction imposed by the compiler:

the compiler, even though it could compile, refuses to compile, when something against the rules

is in the code. But it could compile, but it just refuses to do it.

If it would compile the program anyway, then a runtime error could occur when the type is wrongly

used in the code. You can easily make a program compile with the 'error' in it, by giving the object a generic type.

That takes away the strong typing, which basically only takes away checks, that the compiler performs.

Strong typing is just a helper, not a mandatory thing. Computer systems can function without it.

It is just an extra. It is just something that rules out certain mistakes a programmer can make.

That's the way it is implemented in the new computer language too:

the whole system can function without it, but it extra checking and extra helpers,

imposed afterwards.

Not only the imposing of the rules is very important about type checking.

Also the helpers for the programmer are a blessing. Without any documentation,

autocomplete lists already give an interface to the format an object is expected to have.

Yes, objects have a format. You may only see members in the autocomplete list,

but internally objects just have a format.

That's what a class is: a format. Furtherly, the class is extended with the methods you

could operate on the object of a class.

Enough for now.

JJ

## More from the original Symbol documentation

#### Old

< Also look at Access Media : Interface Access >

> 2009-06-28 later also called Access Ways and Special Access

<<

Does genericity control what happens on the call of system procedures or if the call to a system procedure is illegal or not?

>>

<< Introduce more system procedures here:

Type Generic Get

Type Generic Set

Interface Generic Get

Interface Generic Set

Procedure Reference Generic Get

Procedure Reference Generic Set

Here and no earlier

>>

<< is that the same or different from being able to change type at creation or assignment of object reference ?>>

<<

Old:

A symbol’s type or interface can be generic or explicit.

Generic:

* Generic type: a symbol always adopts the type of its object.
* Generic interface: the interface is the default interface of the type

The type of the object may be generic too. The interface of the type might be generic too. <…>

Explicit:

* The type of the symbol is defined and must match with the type of the object
* The interface of the symbol is defined and must be supported by its type

-----

Assignment can occur by two means: explicit assignment and generic assignment.

The type of the assigned object may be generic too. The interface of the assigned type might be generic too. But the tied together symbols change simultaniously.

Explicit:

* Explicit type:
  + When you assign the object:

It must have this type or an exception occurs

* + When you assign the interface:

The type must support the interface or an exception occurs

* Explicit interface:
  + When you assign the type:

The type must support the interface or an exception occurs

* + When you assign the object:

The object must be of a type that supports the interface (?)

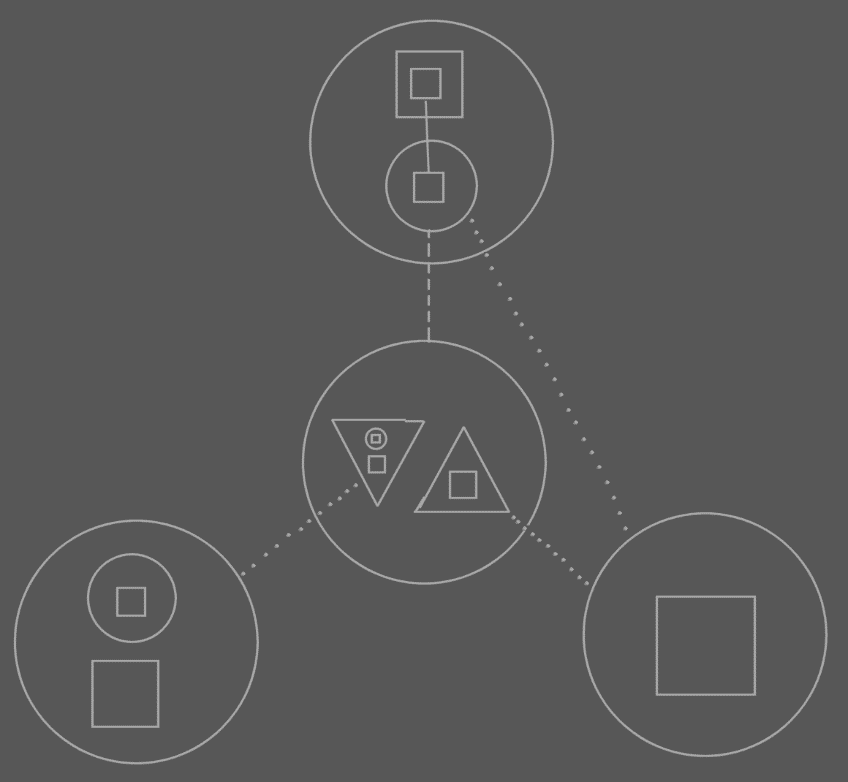
You can change whether or not a type or interface is generic. It is a separate aspect of the symbol. Genericity of type doesn’t mean that it’s of type Object or something, because whenever a symbol has an object, it has a specific type: the type of the object. Whether a symbol has a generic type or interface or not can be set as runtime. Except when the type has been made constant on that part: always a generic interface or always an explicit interface, always a generic type or always an explicit type. Always an explicit type does not mean that the type is constant. The type just always needs to be assigned explicitly, but *can* vary.

>>

If a symbol already has a type line it has to have that type. If it doesn’t have a type line then it gets the type of the object it’s pointing at (not considering the interface problem below)

##### Interface <>

Something like this should be possible:



The point is in the lines of the child of the top symbol. It’s type line points to the center symbol, but it’s interface line points to the interface of ONE of its types interface implementations, not to the joint interface implementation of the type. You have to be able to do this.

But it goes against rules I set up earlier. So there has to be more flexibility with interfaces.

The type line no longer determines the interface line.

Ah, more rules need to be invented for working with interfaces.

The type line is pointing at the center circle, but implicitly points at one of the right triangle. So there’s an invisible extension to the type line. The extension changes with the interface line of the source symbol. However… the interface of the child determines which interface is implicitly pointed at by the type line. This seems an explicit/implicit subject.

Another direction rule can be added I guess, looking at the implicit piece of the type line: Sooner to exit from a circle as line source than to exit from a triangle as the line source. <that rule needs some reformulating, though.>

The rule about the type line determines the interface line stays. It’s just that there’s an implicit extension to the type line, which can be generic, pointing at the interface that is the interface of the child…

##### Type

If you redirect with an object line then the object reference gets the same type as the target object (except when you use a different interface, the previous problem I introduced). Not all the symbols will get a line to the type, though.

##### Operators

|  |  |
| --- | --- |
| Type Generic Get: | ~:: |
| Type Generic Set: | ~:: = |
| Interface Generic Get: | ~: |
| Interface Generic Set: | ~: = |

<< Maybe I’ll do special operator symbols in text code that represent the object relations. Actually, that’s what I’d like a little. Except member access. I wanna keep that. >>

Oh, making type unsettable at run time can be considered a form of genericity. Maybe I should even rename the other type genericity I mentioned. It might be called object-type genericity. Well… settability can also be called variable and constant.

See to it you mention that an explicit type does not fix the type permanently unless you make the Type Set of the symbol Runtime Inaccessible. Oh, haven’t covered access control in detail yet, but that’s not a problem really.