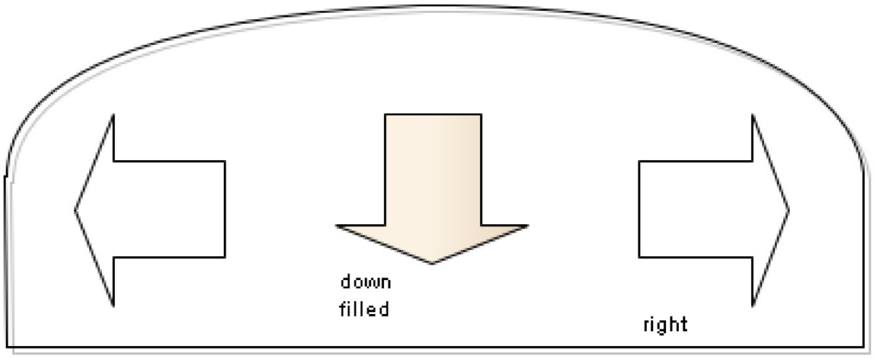
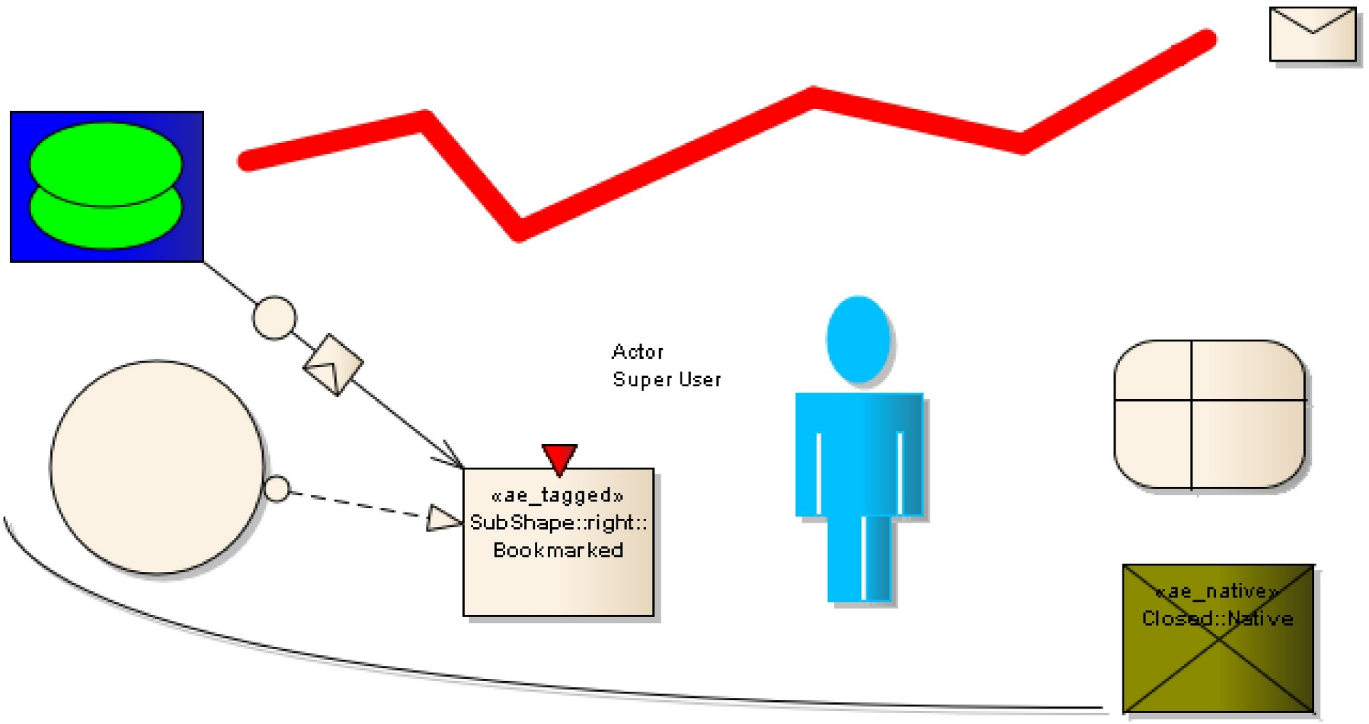
Shape Script

made easy





Drawing individual element shapes with Enterprise Architect

By Thomas Kilian

Shape Script made easy

Drawing individual element shapes with Enterprise Architect

Thomas Kilian

This book is for sale at http://leanpub.com/shapescript

This version was published on 2021-04-06

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Preface

Sometimes – or even often – you need different shapes than those from standard UML. That is something resembling a technical device rather than a rectangle or a stickman. In that case Enterprise Architect[1] offers a nice feature which is called Shape Script. As the name suggest you will find a scripting language which allows you to define the shape of elements and even connectors.

This book is intended as tutorial and reference for the Shape Script language. It offers a step by step introduction, a lot of examples and quite some tricks you need to know when using Shape Scripts.

> ⚠ While assembling this book I took a tour through a lot of corners which seemingly had not been visited by other adventurers. I do not dare to ask whether Sparx has a testing team at all. I marked those strange places with YAEAB (Yet Another EA Bug).

> ⚠ Although Shape Script allows for quite complex graphic caprioles you are strongly encouraged to stick to the simplest possible minimum. If possible do not use it at all (sic!), since standard UML is understood by more people than individual characteristics possibly produced by Shape Script. Dadaism like that on the cover page is possible but in most cases non-sense. However, if you are forced to write your own shapes I hope this book will help you to get it done quickly.

> ⚠ And one warning on top: Shape Scripts are interpreted while EA renders a diagram. So for every shape that needs to be drawn EA will look into the script and do some magic. Depending on the number of elements and the comlexity of the Shape Scripts the rendering time for a diagram might increase significantly. The only ways out are to either use less or no Shape Scripts or to reduce the number of elements that appear on a single diagram.

Anyhow, a common use of Shape Script is in combination with MDG Technology files. Any stereotypes defined therein come along with a number of stereotype properties (aka. tagged values). These can be used to show different shapes and/or text in the element. Quite some MDGs being delivered with Enterprise Architect use this technique.

[1] The EA version used to create this book was actually 10.0 (build 1009). However, most of the references are also valid for earlier versions of EA. And of course all this still works with version 13.5 when the latest edit was made on this book. Though I have not cross checked which bugs were fixed until 13.5. Honestly, V14 itself was so buggy I never used it. I might cross check with V15.2 but I'm not to keen installing newer EA versions.

2. Copyright and Disclaimer

Also all of the information in this book has been tested by me in many circumstances I can not hold any liability for use of the here presented information[1]. However, I'd be glad to receive any kind of feedback to correct future updates of this book which you will receive for free in turn. Having said this, all information presented here is subject to change without notice.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

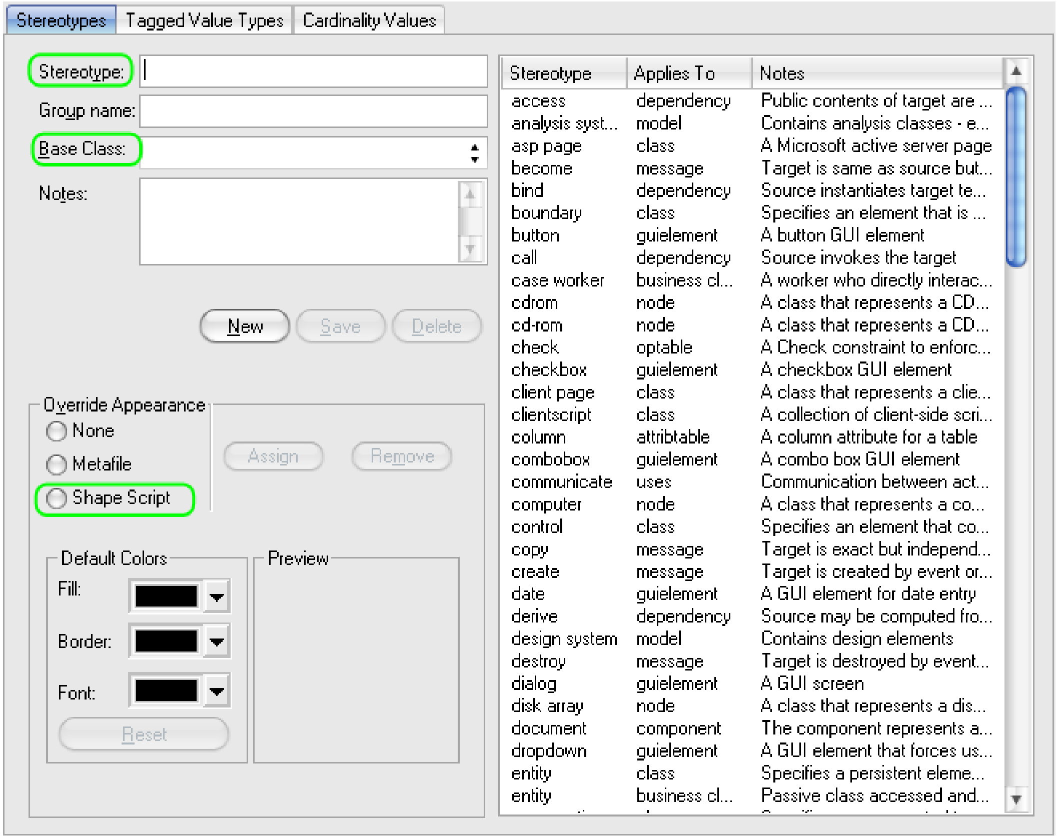
[1] I really loathe writing such legal blurb since it should be obvious. By the way: German Law applies! (Does that change anything?)

3. Basic Concept

Shape Scripts in Enterprise Architect are a way to assign individual shapes to stereotyped elements and connectors. The language is somewhat C-like but is limited to quite a small subset of control structures and it offers a number of build-in graphic drawing methods. The advantage is that you can learn it rather fast. The drawback is that you soon reach limits when trying to do more advanced graphics. This however should be acceptable as elements should not be complex graphic art but meaningful symbols.

We will start with shapes for elements before explaining connectors. An element shape has a size of 100 units in width and height. The top left (X|Y) coordinate is (0|0) and the bottom right coordinate is (100|100). These coordinates will be scaled to whatever you size the element on a diagram.

Before going into details of the language itself let us try a simple example. Open `Settings/UML Types/Stereotypes` in any temporary Enterprise Architect repository you would like to use as sandbox.



Stereotype	Applies To	Notes
access	dependency	Public contents of target are ...
analysis syst...	model	Contains analysis classes - e...
asp page	class	A Microsoft active server page
become	message	Target is same as source but...
bind	dependency	Source instantiates target te...
boundary	class	Specifies an element that is ...
button	guielement	A button GUI element
call	dependency	Source invokes the target
case worker	business cl...	A worker who directly interac...
cdrom	node	A class that represents a CD...
cd-rom	node	A class that represents a CD...
check	optable	A Check constraint to enforc...
checkbox	guielement	A checkbox GUI element
client page	class	A class that represents a clie...
clientscript	class	A collection of client-side scri...
column	attribtable	A column attribute for a table
combobox	guielement	A combo box GUI element
communicate	uses	Communication between act...
computer	node	A class that represents a co...
control	class	Specifies an element that co...
copy	message	Target is exact but independ...
create	message	Target is created by event or...
date	guielement	A GUI element for date entry
derive	dependency	Source may be computed fro...
design system	model	Contains design elements
destroy	message	Target is destroyed by event...
dialog	guielement	A GUI screen
disk array	node	A class that represents a dis...
document	component	The component represents a...
dropdown	guielement	A GUI element that forces us...
entity	class	Specifies a persistent eleme...
entity	business cl...	Passive class accessed and...

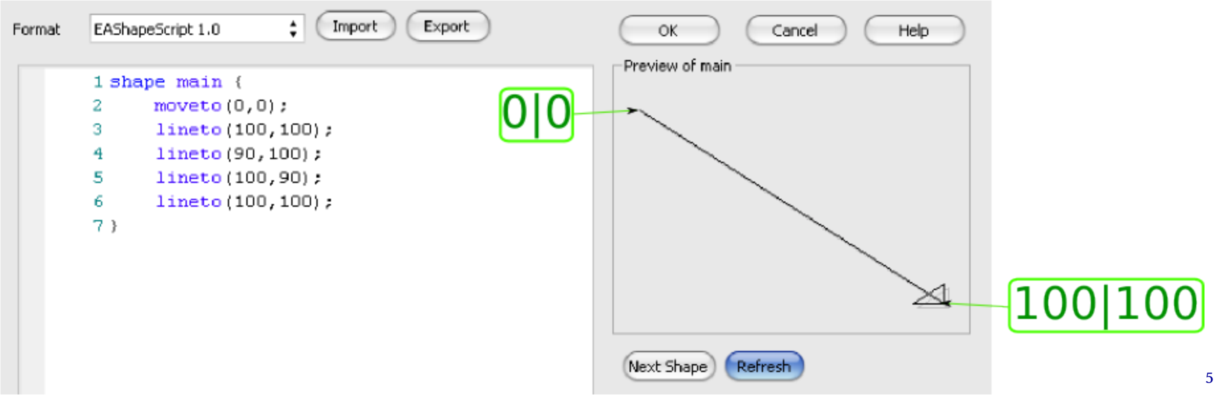
Then create a new stereotype by entering `ae`[1] in the `Stereotype` field and choosing `<all>`[2] from

[1] This name will appear on top of the existing list of stereotypes so you can edit your test cases faster than with one named `test`. I use `ae` as prefix for element stereotypes and `ac` for connector shapes in this book. You are asked to use meaningful names for you stereotypes instead.

[2] For a real case you should limit it it the base element where you want to appear the stereotype.

the `Base Class` drop down. That will allow to assign the stereotype to all elements and see what the shape will look like for them.

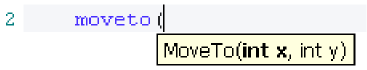
Now the important part: check the `Shape Script` radio button and click `Assign`[3]. This will open the Shape Script editor where you can type the script in the left pane. By clicking `Refresh` you will see the resulting image in the right one[4].



```
shape main {
    moveto(0,0);
    lineto(100,100);
    lineto(90,100);
    lineto(100,90);
    lineto(100,100);
}
```

[5]

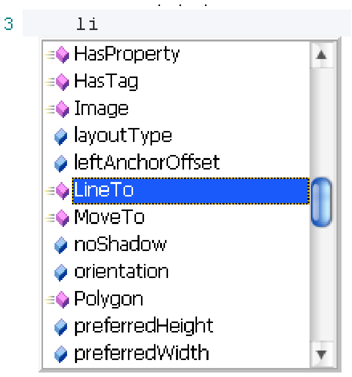
While typing the opening bracket after the `moveto` and `lineto` you will notice that the parameter list for the method is shown as balloon help.



```
moveto (
```

MoveTo(**int x**, int y)

Further when hitting `Ctrl-Space` at any time the editor will open a drop down with possible methods starting with the characters already typed for the current word (top of the list if at new line or at a space):



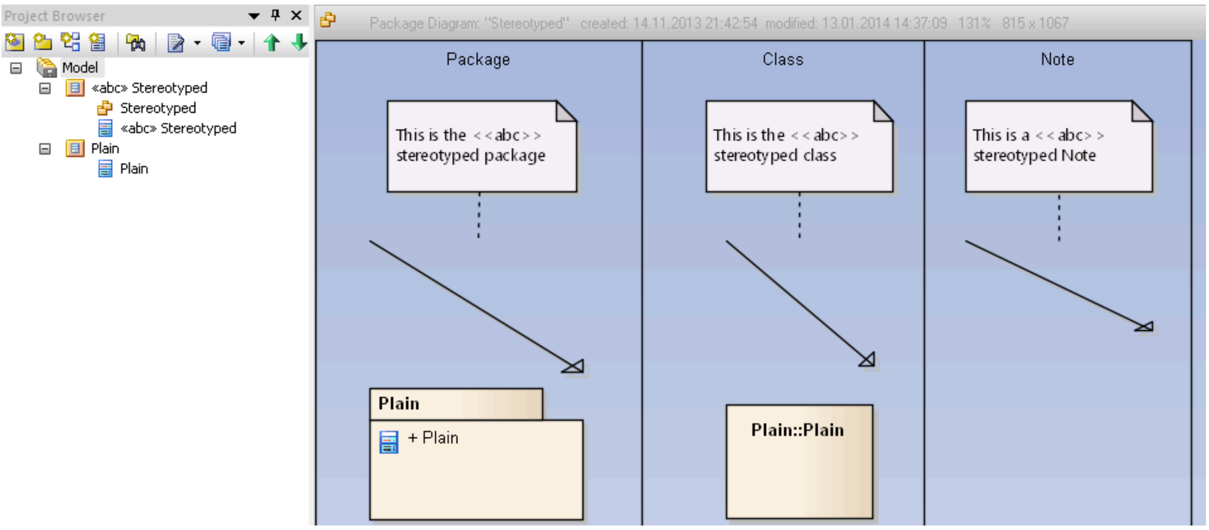
```
li
```

Once you have saved the script in the editor **and** the «ae» stereotype as such you may apply the stereotype to see how it looks like on a real diagram. This may differ from the preview in certain cases. Also you can test scaling and later then conditional drawings.

[3] Once a script is assigned the button will appear as `Edit` and the `Preview` below will contain what it says.

[4] The `Next Shape` button will only be relevant if there are more than one shapes defined in a script. The preview will simply loop through the shapes and display one after the other.

[5] I highlighted the top left and bottom right coordinates in the screen shot.



This is the <<abc>> stereotyped package

This is the <<abc>> stereotyped class

This is a <<abc>> stereotyped Note

The result looks a bit dull. But it shows the basic principle how to assign and test Shape Scripts.

3.1 Control Structures

As already mentioned Shape Script is very limited in its capabilities. So are control structures. You only have `if` and `else` to control the flow of statements. There are no loops at all.

The format for that is:

`if (<query>) <block-or-statement>`

where `<query>` is one of the methods described below. `<block-or-statement>` is either a sequence of the graphic methods described above enclosed in { and }-braces or a single statement . And of course any `if`-construct counts as statement as well.

Yes. There is no comparison or boolean expression. Just those methods.

`if (<query>) <block-or-statement> else <block-or-statement>`

is hopefully obvious. Syntactically you can abbreviate `if`-cascades by using `if (<query>) <block-or-statement> else if (<query>) <block-or-statement> ...`

`return`

is a single statement that may appear at any position. It will stop further processing of the Shape Script immediately. You can use this statement to mimic a `case` statement instead of using an `if`-cascade.

3.2 Query Methods

The use of queries is a more advanced[6] feature and will be used later in chapter Advanced Usage. So here's just a general overview of the operations.

[6] Not so say "the most advanced feature in Shape Script".

`HasTag(tagName)`
: will evaluate `true` if the tagged value named `tagName` exists at all.

`HasTag(tagName,tagValue)`
: will evaluate `true` if the tagged value named `tagName` exists and has a value equal to `tagValue`.

⚠ If you have multiple tagged values with the same name (or even mixed case) this method will take an arbitrary tagged value for evaluation.

The values for `property` in the two operations below are explained in chapter Properties. This and the parameter `value` must be supplied as string, i.e. enclosed in either single or double quotes.

`HasProperty(<property>)`
: will evaluate `true` if the property named `property` exists at all. E.g. `HasProperty("alias");` will evaluate `true` only of a alias has been defined in the properties.

`HasProperty(<property>,<value>)`
: is the same as the previous method except that it checks for equivalence of `value` and the result of `property`. So you could check if an element is named specifically (which only makes limited sense).

ℹ The comparison is case insensitive. That is true for both `property` and `value`.

4. Shaping Elements

As already mentioned the Shape Script language is a bit C-like. So probably most people will not have much trouble to learn the syntax. Anyway it's very limited. An EBNF syntax description can be found in the appendix.

Generally all keywords and even strings are case insensitive. So it does not matter whether you write `LineTo` rather than `lineto` in the above example. The auto-completion suggests the first variant in camel case which is definitely better to read.

4.1 The Main Shape

Or to talk Sparxian: `shape main`. As you already have seen, these two keywords introduce the body wrapped in curly brackets. The instructions inside will be executed each time a stereotyped element is shown on a diagram. As already mentioned each element has 100 units in width and depth (as opposed to height since the units increase downwards). Even shapes which appear oval (like Use Case) have that rectangular 100^2 units frame.

> Connectors always attach to that drawing frame and not to something which is drawn inside. You will probably have seen that already with Use Case elements.

Now let's see what can be done to actually draw something. Let's start with the two methods `LineTo` and `MoveTo` used in the introductory example.

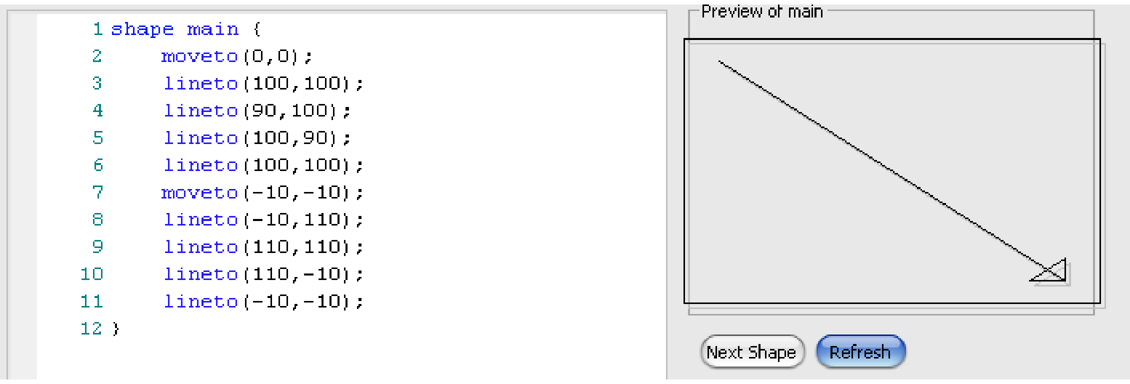
Simple Lines

There is not much to say:

`MoveTo(x,y)`
: moves the graphic cursor to the specified coordinate. Initially in the script the cursor is located at (0|0).

`LineTo(x,y)`
: draws a line from the current coordinate to the new `(x|y)` and sets the current cursor to that position.

> Actually you can not only draw inside the drawing frame. You can extend it at your wish as the following example shows. Results might be funny as the Preview pane shows. Just try what happens if you extend the coordinates even more.



```
shape main {
    moveto(0,0);
    lineto(100,100);
    lineto(90,100);
    lineto(100,90);
    lineto(100,100);
    moveto(-10,-10);
    lineto(-10,110);
    lineto(110,110);
    lineto(110,-10);
    lineto(-10,-10);
}
```

... omitted ...

5. Shaping Connectors

Basically the shape for connectors will be defined similarly to that for elements. So most of the previously explained methods can be used for connectors too (of course some – like compartments – do not make sense).

A major difference between both shapes is that elements all have that 100^2 unit frame. Connectors are not that easy. Shape Script distinguishes between different parts of a connector: source, target, the main connector line and the six labels. For each of them you can define a `shape` routine:

`shape main { <block> }`
will define what appears for the connector line.

`shape source { <block> }`
will define an extra shape at the source end of the connector.

`shape target { <block> }`
will define an extra shape at the target end of the connector.

`shape <labeltype> { <block> }`
will draw the `<labeltype>` labels according to the statements in `<block>`. Here `<labeltype>` is one of `LeftTopLabel`, `MiddleTopLabel`, `RightTopLabel`, `LeftBottomLabel`, `MiddleBottomLabel` and `RightBottomLabel`.

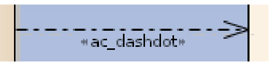
Now let's see what happens when we put some code in these shapes.

5.1 The Connector

We will start with a simple example: using a dash-dotted line which – if my memory does not deceive me – is not used in UML. The following Shape Script is assigned to the base class dependency.

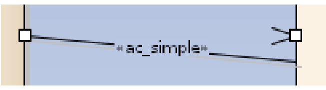
```
shape main {
  SetLineStyle("dashdot");
  MoveTo(0,0);
  LineTo(100,0);
}
```

When used it will produce the following:



That was easy. But what happens when you actually draw something else? Well, let's try by using the whole 100^2 units.

```
shape main {
  MoveTo(0,0);
  LineTo(100,100);
}
```



... omitted ...

6. Properties

As mentioned earlied, Shape Script makes use of a number of properties which can either be used in `Print` statements enclosed in `#` (see chapter Strings) or as parameter in the `HasProperty` operations (see chapter Query Methods).

EA's documentation of the possible properties leaves some room for improvements - or speculations. So I went through the list and looked for the results found in diagrams when applied.

Basically the properties are strings - and most of them work as expected. While all (well almost) of the strings can be used in a `Print` statement, their use in `HasProperty`is partially restricted to either element or connector shapes.

6.1 Element properties

Trivial

These properties just relate to the ones matching the element properties dialog: `alias, author, complexity, datecreated, datemodified, keywords, language, metatype` (like defined in a MDG), `name, notes, scope, status, stereotype, type` (e.g. "Class")

The two `date` properties are pointless as you can only test for in-/equality in Shape Script. The format to test is exactly that what you copy from the properties. In my localization it's "DD.MM.YYYY hh:mm:ss" and I wonder how the guys over the Atlantic ocean will see this to work commonly. I guess it won't work.

… omitted …

7. Advanced Usage

Probably you do not want to define Shape Scripts directly in the stereotypes. More likely you are going to deploy them along with an MDG Technology file. I can not explain how to create MDGs in this book as it would lead much too far. So you either know already how to do that, or you need some outside help.

Another advanced usage is when you simply need more than Shape Script can deliver in respect to querying model contexts. That is where add-ins come into play.

7.1 Shape Script in MDG

Any Shape Script for a profile element must be defined by adding a property named `_image` and clicking the ellipsis button next to the `Initial Value` field. This will open the Shape Script editor.

> It is advisable to first test the shape using the method described in the beginning of this book. Else you would need to go through creating and deploying the MDG before you can see what was wrong with your Shape Script.

7.2 Add-in

Since Shape Script is so limited in performing algorithms there is an escape through the use of external code hosted in an add-in. If you want to use this feature you need to know how to write add-ins at all. I can not explain how to do that so you need to get outside help for that. But if you know it then here is what you need to take advantage of this escape.

Basically you can retrieve a string value from your add-in which you can evaluate by `HasProperty` or by directly printing it using a properties. The format is

`addin:<addin_name>, <function_name> {, <parameter>}`

where `<addin_name>` is the name you had chosen for your add-in (the identifier) and `<function_name>` is the name of the function inside your add-in. An arbitrary list of comma separated parameters can be supplied which are passed by value to the called add-in procedure where the repository, the element-GUID and the additional parameters[1] are passed. Since Shape Script knows neither variables nor string substitution you need to write those by hand in any case. So a single parameter will usually suffice – or you just have named functions. The called function must return a string as result.

E.g.

[1] Since I use Perl, I only can see the repository and the GUID parameter so I have to use unique functions. This is perfect anyhow as these add-in methods should be used in rare cases only.

```
print("#addin:myAddIn,pFunc1#")
```

will print the result returned by the function `pFunc1` inside your add-in framework. Similarly

```
hasproperty('addin:myAddIn,pFunc2', '1')
```

will evaluate to `true` if your function `pFunc2` returns the string value `1`.

Of course there's also a major drawback here. Since the anyway interpreted Shape Scripts are relatively slow, each time your add-in is called will be on top of the rendering time. So whatever you are doing in your add-in: it should be done quickly.

8. Advanced Patterns

This chapter simply shows a couple of shapes you could adapt for your own use. Currently they are not ordered in any way.

And if the following are not enough you should visit Geert Bellekens'[1] public repository at https://github.com/GeertBellekens/Enterprise-Architect-Shapescript-Library. It contains all disassembled Shape Scripts that Sparx has used in their MDGs. And, not to forget, Sparx has given permission to make them public.

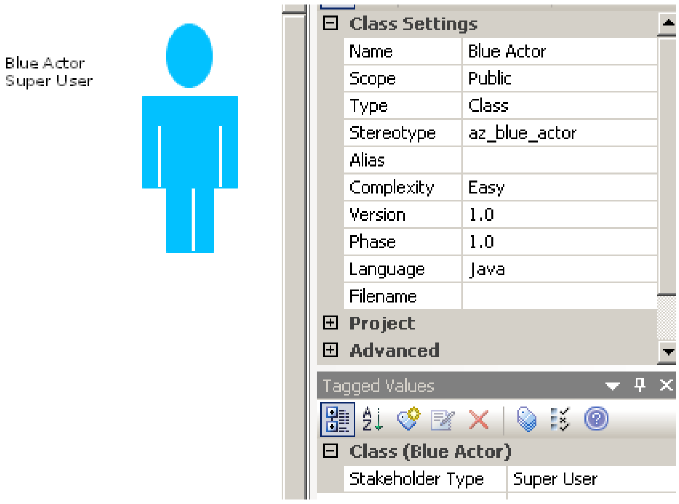
8.1 Different Actor

This nice actor symbol comes courtesy of Andy J (from the Enterprise Architect forum).

```
shape main {
// Blue Person courtesy of Andy J;
  fixedAspectRatio = "true";
  SetFillColor(0,192,255); // light blue
  SetPenColor(0,192,255); // ditto
  Ellipse(70,-15,90,20);  // head
  Rectangle(60,25,100,75); // chest/arms
  Rectangle(70,75,90,110); // legs
  SetFillColor(255,255,255); // white "shadows"
  Rectangle(80,75,82,110); // legs
  Rectangle(66,40,68,75); // left arm
  Rectangle(92,40,94,75); // right arm
  Println("#NAME#"); // name it
  Println("#TAG:Stakeholder Type#"); // add. tag info
}
```

displays as

[1] https://bellekens.com



⊟ Class Settings

Name	Blue Actor
Scope	Public
Type	Class
Stereotype	az_blue_actor
Alias	
Complexity	Easy
Version	1.0
Phase	1.0
Language	Java
Filename	

⊞ Project

⊞ Advanced

Tagged Values

⊟ Class (Blue Actor)

Stakeholder Type	Super User

8.2 Composite Symbol

In case you want to show the lying 8 (composite symbol) on your shape you simply need to add the following at the end of your script:

```
decoration composite {
  orientation="SE";
  if(HasProperty("iscomposite","true")) {
    Ellipse(-80,25,-10,75);
    Ellipse(10,25,80,75);
    MoveTo(-10,50);
    LineTo(10,50);
  }
}
```

If you already have a decoration at the SE position you need to merge the above code, not simply add the decoration as there can be only one decoration at SE.

8.3 Non-/Rectangular Notation

If you want the user to allow switching between rectangular and iconic representation you simply can do that with

```
if (HasProperty('rectanglenotation', '0')) {
  // code for iconic representation
) else {
  // code for rectangular representation
  // e.g. Rectangle or DrawNativeShape
}
```

If the shape script finds this query it will show the `Advanced/Use Rectangular Notation` in the context menu.

⚠ This will not work for the base classes `UseCase` and `Actor` where the context menu option `Use Rectangular Notation` will always appear greyed out. It will however work with the base classes `Class`, `Action` and `Activity`.

ℹ The checkmark indicating the use of rectangular notation seems to be inverted. EAUI.

ℹ Besides the `Use Rectangular Notation` in the `Advanced` context menu you can add your own options there. The way to do it (via MDG) is summarized on one of Sparx' help pages[2].

... omitted ...

[2] http://sparxsystems.com/enterprise_architect_user_guide/12.1/building_models/query_methods.html

9. Shape Script Syntax

This is the EBNF for Enterprise Architect's Shape Script language. The start symbol is ShapeScript. Any spaces and tabs between non-terminals are ignored.

ShapeScript = { Shape | Decoration };

Shape = "shape"[1] ShapeName ShapeBody;

ShapeName = /* any reserved or non-reserved string literal depending on the context */;

Decoration = "decoration" Name ShapeBody;

Name = /* an arbitrary string that should describe the form of the decoration */;

ShapeBody = "{" {InitializationAttributeAssignment} {DrawingStatement} {SubShape} "}"

SubShape = Shape /* with a non-reserved name */;

InitializationAttributeAssignment = Attribute "=" Value ";";

Attribute = /* see chapter Shape Attributes for a list of values /*;

Value = StringLiteral | Integer | Tuple;

StringLiteral = Quote { Character } Quote;

Quote = /* the double quote " or a single quote ' */;

Character = /* any printable character except the used Quote */;

Integer = ["-"] {"0" .. "9"};

Tuple = "(" Integer "," Integer ")";

Block = "{" {DrawingStatement} "}" | DrawingStatement;

DrawingStatement = IfElseSection | Method;

IfElseSection = "if" "(" QueryExpression ")" Block ["else" Block];

QueryExpression = /* see chapter Query Methods for the 2 methods and their parameters */;

Method = /* see chapter Shaping Elements for possible methods and their parameters */;

[1] Actually you can arbitrarily replace "shape" with "label" and "text". You may do so to confuse others.