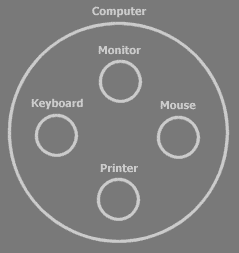
|  |
| --- |
| Circle Language Spec |

## Introduction

This chapter aims to introduce the basics of Circle language. Circle language is an idea for a computer programming language, that is mostly a diagram notation for visualizing computer code.

### Splitting up Ideas

Something that might play a role in making software, is splitting up a larger idea into smaller ideas.



Here you can see the idea of *the computer*, split up into four sub ideas: the monitor, keyboard, mouse and printer.

In Circle language, the main symbol for an idea might be a *circle*:

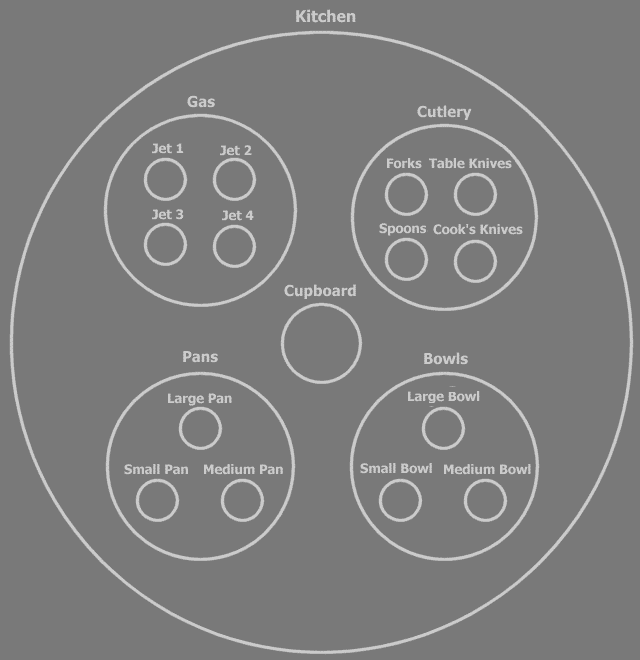


Each sub idea could work more or less independently of the othes. **Monitor** does something, **Keyboard** does something, and so on. It's the larger idea that could tie the sub ideas together. Meaning: the computer could make the link between monitor, keyboard, mouse and printer. The larger idea might combine the sub ideas and could manage the communication between the sub ideas.

In computer programming, ideas, so both larger ideas and sub ideas, might be called *objects*. Each object can be responsible for its own part of the system.



Each sub idea can be split up in sub ideas itself and you can go on and on splitting up ideas.



But to see the general point of the system, you might only need to see the top layers of the split up.

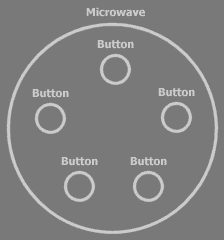
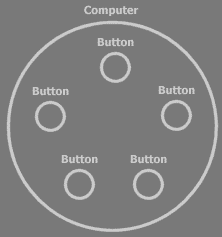


This can make object oriented programming an nice way to keep overview over a system as it grows. The understandability of the design could be helped by a clear the split up into ideas is, perhaps the top layers most of all. Depending on who you ask, a subdivision in objects might be the most important thing in software. So splitting up an idea into sub ideas, something that most people are probably able to do, might actually be the most important thing about programming.

### Reusing Ideas

Good design in grouping ideas can lead to the *reusability* of ideas. For instance: the idea of a *button* might be reusable. It’s used everywhere. If you would program the perfect button, nobody would ever need to program a button again. You could just reuse the same idea whenever you need a button.

Any place you use a button in your system, there is a separate object:

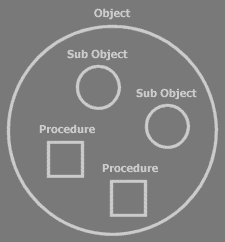
 

A button object in general is a *type* of object. All the button objects are objects of the same *type*. The separate buttons are so called *instances* of the type. Many times the word *object* is used instead of the word *type*. For instance, you can speak of *the* button *object*, while you’re talking about the button *type*. It’s like by saying ‘panda’ you can refer to an individual panda (object), but also to the panda as a kind of animal (type).

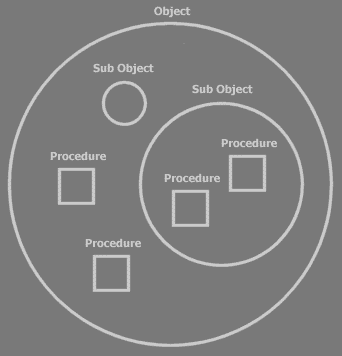
There are many types already defined, that you will be using one way or another. Such as the **Button** type for instance, which you can use to quickly build a simple user interface. Another type, **Integer**, represents one integer number. You can hardly avoid using **Integers**.

### Procedures

Apart from a collection of sub objects, an object contains *procedures*.

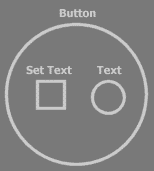


Sub objects again contain procedures.



Software can’t execute without procedures. When you run a procedure, an object *does something*.

A button could have a **Set Text** procedure for instance, which sets the text displayed on the button.

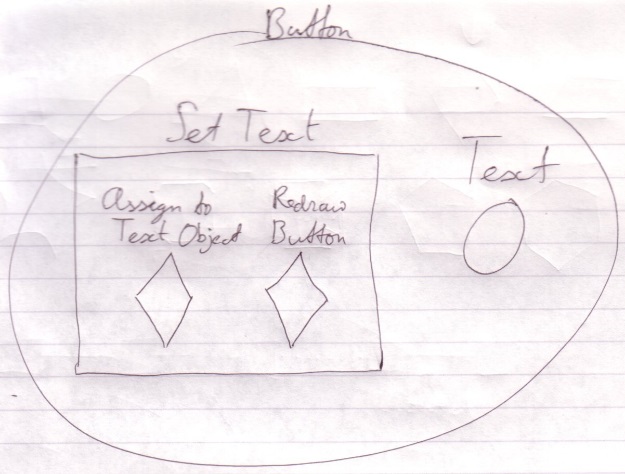


Procedures are denoted by *squares*.

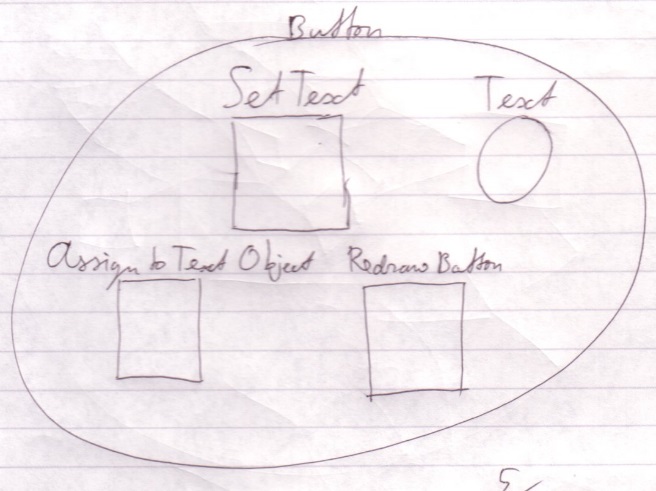
Other objects could also have a **Set Text** procedure, therefore to identify the **Set Text** procedure of the ***Button*** object you notate:

Button . Set Text

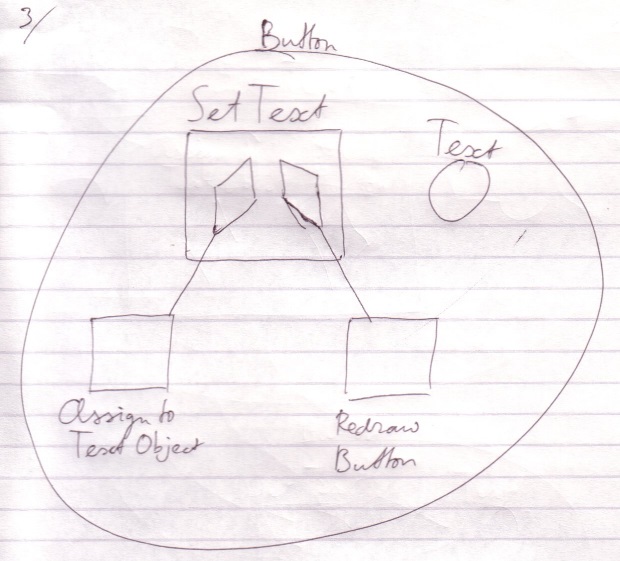
A procedure itself consists of sub procedures: the separate steps of the procedures.



Sub procedures are usually not embedded inside other procedures. They are defined separately:

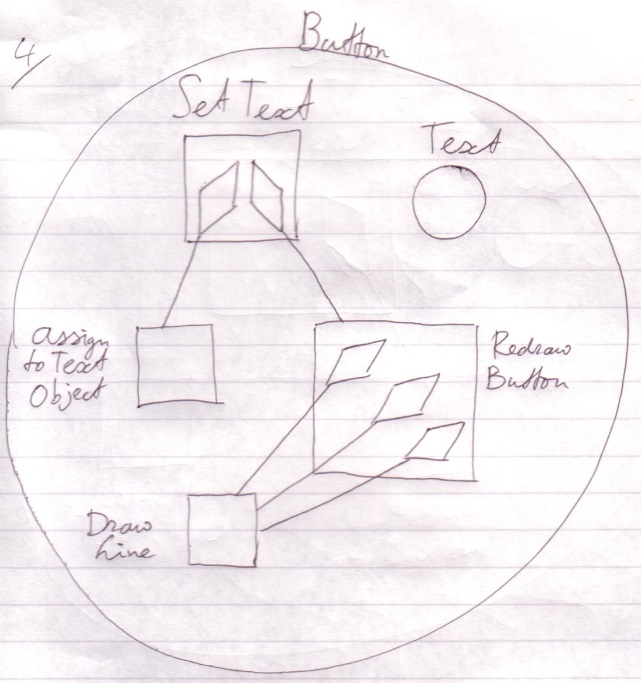


And the sub procedures are *called* from the super procedure:



Calling a procedure has almost the same effect as inserting the called procedure there where you call it.

The **Redraw Button** procedure is rather complex. In this procedure lines of the button are drawn, the text is drawn and whatever else makes up the display of the button. The steps of this redrawing are delegated by yet again calling other procedures. **Redraw Button** could for instance call the **Draw Line** procedure a number of times. So sub procedures of the **Set Text** procedure are themselves composed of sub procedures.

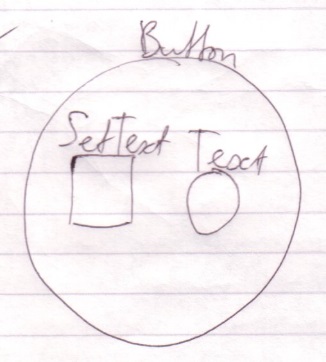


You can also see here that separately defining a procedure can lead to the *reuse* of procedures. The **Draw Line** procedure is reused three times in the example above. There are many procedures already defined, that you will use one way or another. Most of these procedures are encapsulated in a type. There’s for instance a **File** type that contains procedures with which to control a single computer file.

So where does it end? Procedures would continue to delegate to one another and nothing would really actually happen. Well, it ends at a special group of procedures that don’t call other procedures anymore. Each of those procedures executes a so called machine instruction: a basic instruction that is sent to the computer’s central processing unit (CPU) and make the *computer* *do* something. The CPU performs a hardware defined *machine* procedure.

That way there can develop a big procedure call tree-out, which makes a single procedure consist of many, many machine instructions, ranging from tens to thousands of machine instructions or even more.

Justs like with the kitchen example: you don’t necessarily need to see the details of the procedure in order to understand what the effect **Button . Set Text** is.

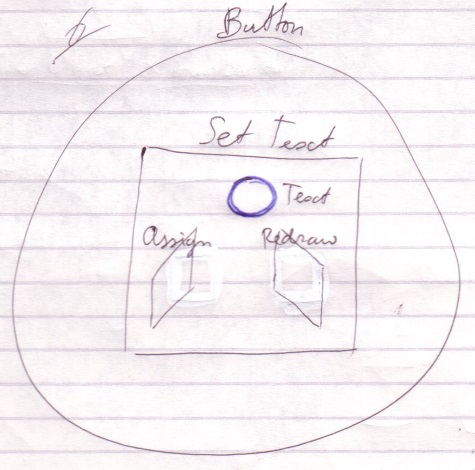


called *operations*, *functions* or *routines* and there are even more synonyms. But I will usually stick to the term *procedure*.

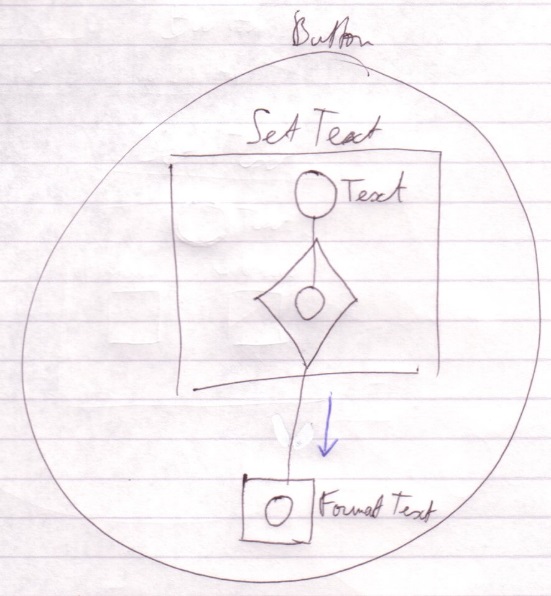
### Procedure Parameters

Procedures can have *parameters*: instructions passed along with the procedure that make the procedure behave differently. The **Button . Set Text** procedure, for instance, has a **Text** parameter, which indicates what the new text of the button will be.

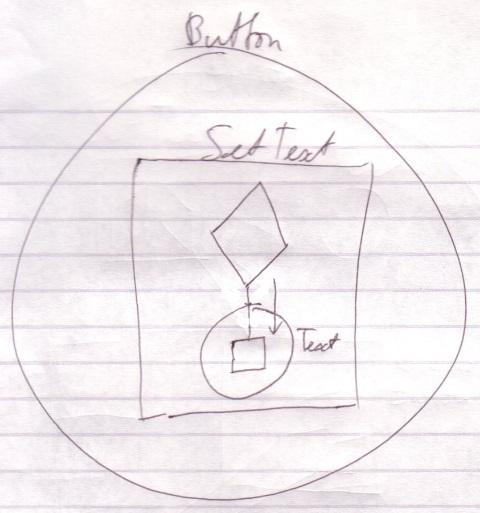
Text is an *idea*. It is an *object*. It is *objects* that serve as parameters.

**

The procedure can do whatever it wants with the parameters passed to it. All it really can do with it is again pass the parameter to another procedure *or* call procedures of the parameter.



*Pass the Text parameter on to the next procedure (Format Text).*

**

*Call the a procedure of the Text parameter.*

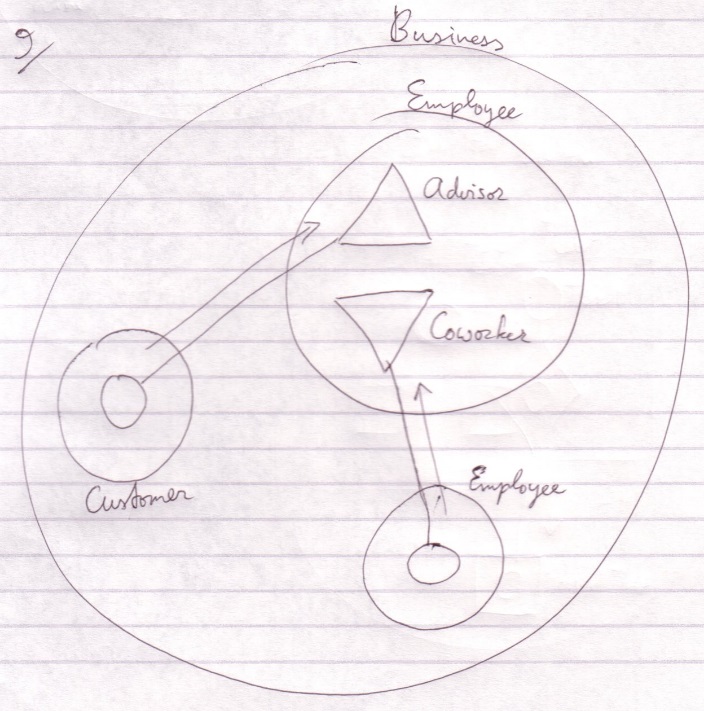
The terms *parameter* and *argument* are often intermixed. For now you can assume that a parameter is a setting of a procedure and an argument is the value that it holds. That’s not the entire truth, though, but I’ll save the exact meaning for later.

### Interfaces

Another issue of objects is that they can have different interfaces. Consider how a employee looks to a customer and how she looks to one of her coworkers. She can do certain things for her coworker and much different things for a customer. She has two interfaces. The party that refers to her determines which interface she has, but even though both parties are referring to the same person, she means two totally different things to them.

Objects have the same ability. A type of object can have separate interfaces. Interfaces are groups of an object’s members. Another object that refers to the object decides which interface to use, or may even only be able to use a certain one of its interfaces. An object can also have just one interface, the same one for everybody.

Interfaces are distinguished by triangles.

**

The employee is an advisor to the customer and to another employee a coworker.

Where to start when you read a diagram like this? You first read the containment structure. You start at the largest container, then the smallers ones. After that you look at the relations made by the connecting lines.

You don’t always start with the largest container. Sometimes one symbol is highlighted. Then this symbol is what the diagram is about and it's all about the aspects of this symbol. Then you start at the highlighted symbol and look around. There’s no one way of doing it.

Triangles are actually sub objects too. The difference with circles is that triangles basically melt together with their container.

Objects, be it triangles, are an excellent way to give different types mutual characteristics.