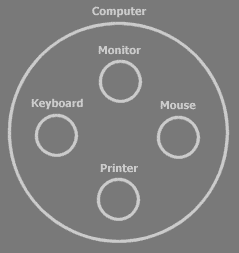
|  |
| --- |
| Circle Language Spec |

## 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 can 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*:

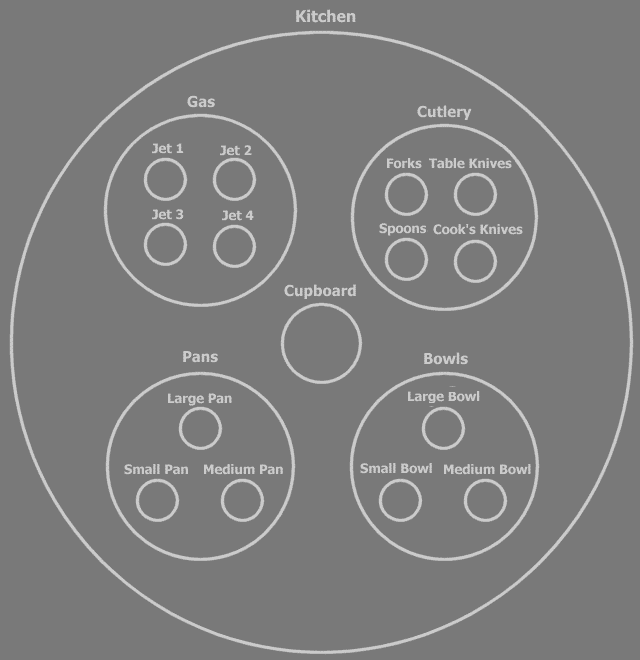


Each sub idea could work more or less independently of the others. **Monitor** does something, **Keyboard** does something. It might be the super idea that would tie the sub ideas together. Meaning: the computer could make the link between monitor, keyboard, mouse and printer. The super idea might combine the sub ideas and could manage the communication between the sub ideas.

In computer programming, ideas, so both super 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 mith 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.

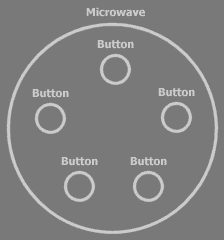


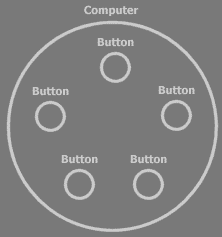
This might make object oriented programming a way to keep overview over a system as it grows. A clear the split up into ideas might help the understandability of the design, perhaps most importantly the top layers. Depending on who you ask, a subdivision in objects might be the most important thing in 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

Dividing up into ideas may lead to the *reusability* of ideas. For instance: the idea of a *button* might be reusable. It could be used everywhere. If you would program the perfect button, nobody might 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 could be a separate object:





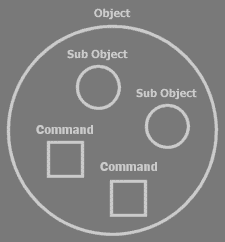
A button object in general can be called a *type* of object. All the button objects can be considered objects of the same *type*. Sometimes the word *object* is used instead of the word *type*. For instance, you might speak of *the* button *object*, while you’re talking about the button *type*. It’s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

There could be many types already defined, that you might be using one way or another. Such as the **Button** type for instance, which you can use to build a user interface. Another type, **Integer**, represents one integer number. You might be using those a lot when you program.

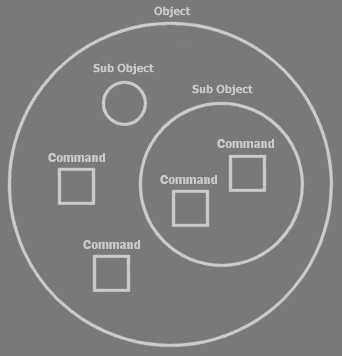
Another word for type is *class*. *Type* and *class* might not be precisely the same things, but it is pretty close and the word *class* is used a lot in several object oriented programming languages.

### Commands

Apart from a collection of sub objects, an object can contain *commands*.

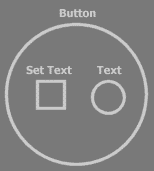


Sub objects can again contain commands.



Software cannot execute without commands. When you run a command, an object *does something*.

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

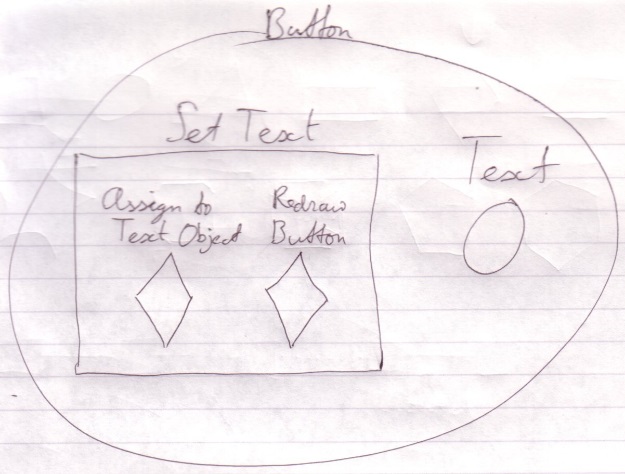


Commands are denoted here by *squares*.

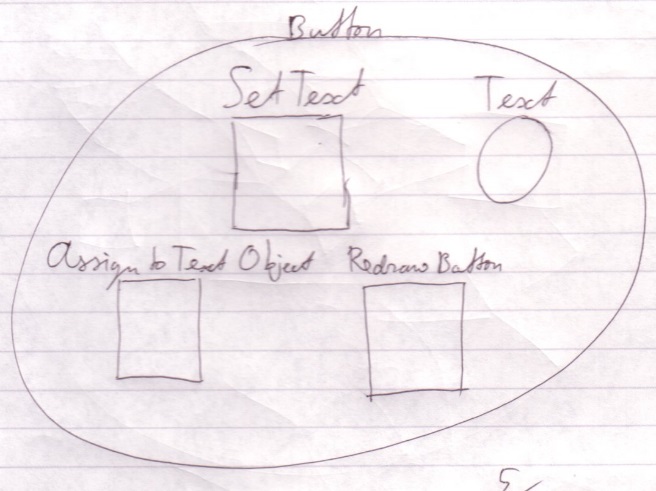
Other objects could also have a **Set Text** command, so to identify the **Set Text** command of the **Button** object, this might also be expressed something like:

**Button . Set Text**

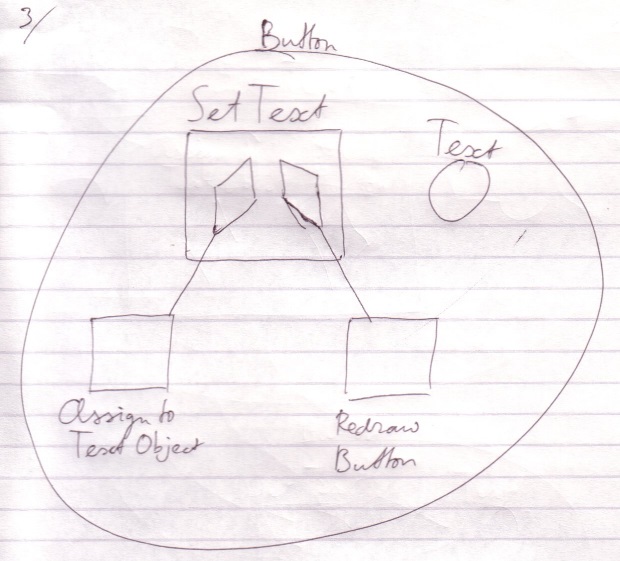
A command itself can consist of sub commands: the separate steps of the command.



Sub commands are not necessarily embedded inside other commands. They can be defined separately:

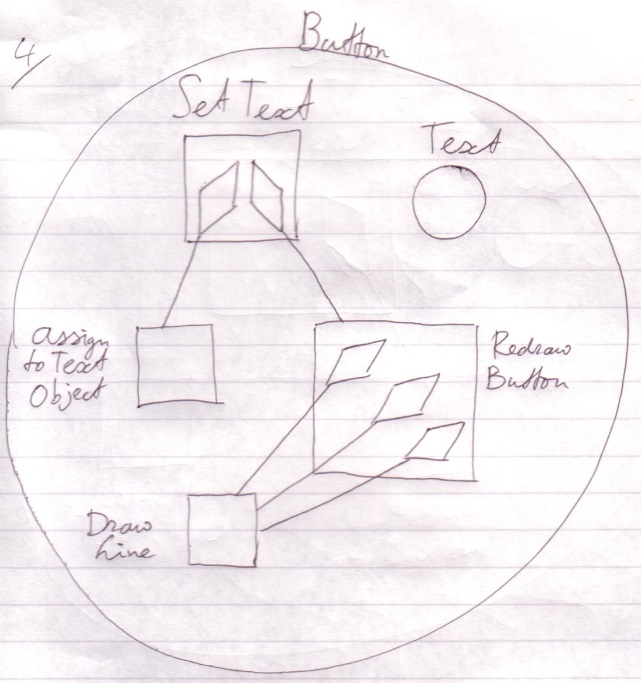


And the sub commands can be *called* from the super command:



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

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

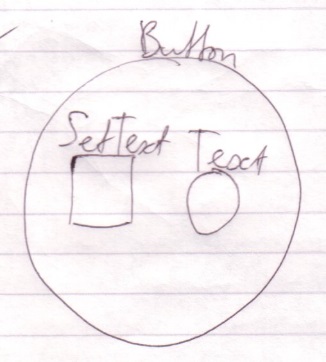


You can also see here that separately defining a command can lead to the *reuse* of commands. The **Draw Line** command is reused three times in the example above. There may be many commands already defined, that you will use one way or another. Commands like that can also be put inside in a type. There could for instance be a **File** type that contains commands with which to control a single computer file.

So where does it end? Would commands continue to delegate to one another and would nothing really actually happen? Well, it seems to end at a special group of commands, that do not call other commands anymore. Each of those command 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 would perform a hardware defined *machine* command.

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

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

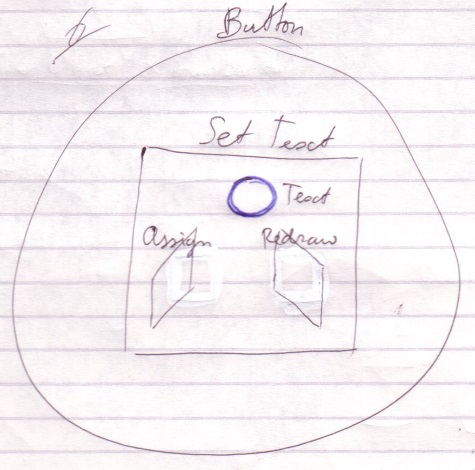


Other names for commands could be *operations, procedures*, *functions, methods* or *routines* and there are even more synonyms.

### Command Parameters

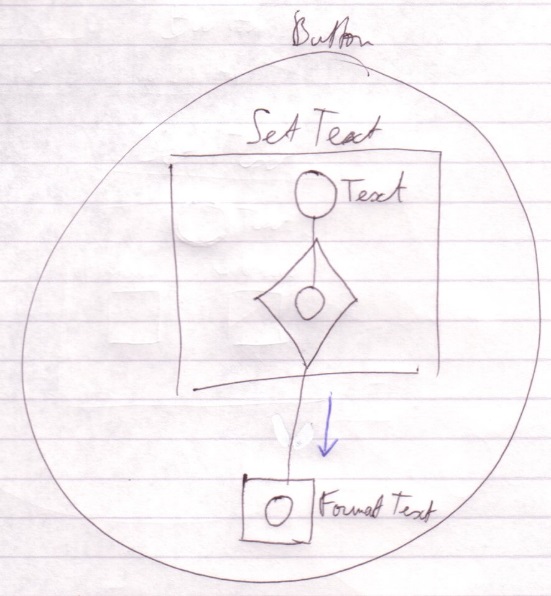
Commands can have *parameters*. Those are like instructions passed along with the command that make the command behave differently. The **Button . Set Text** command, for instance, has a **Text** parameter, which indicates what the new text of the button will be.

Text can also be seen as an *idea*. It can be thought of as an *object*. That way, it is *objects* that serve as parameters.

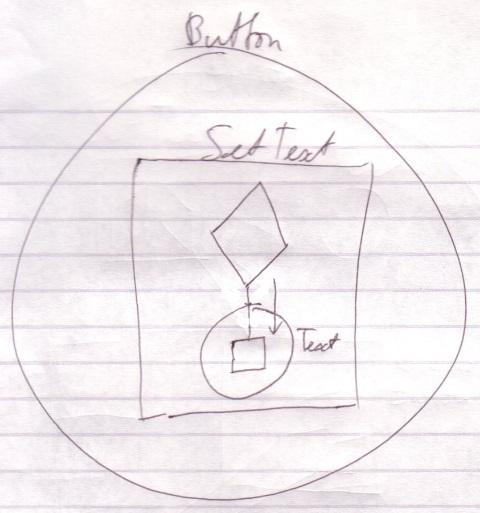
**

The command can do whatever it wants with the parameters passed to it. Arguably, all it really can do with it is:

* again pass the parameter to another command *or*
* call commands of the parameter.



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

**

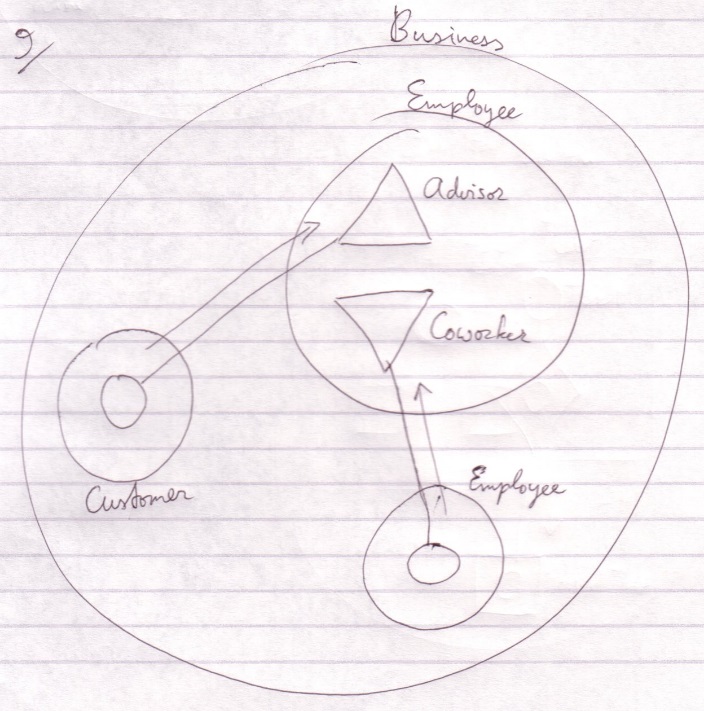
*Call a command of the* ***Text*** *parameter.*

The terms *parameter* and *argument* are sometimes used interchangedly. They are not entirely the same thing, but the difference might be too suble to talk about right now.

### Interfaces

Another issue of objects is that they can have different interfaces. Try to 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 different things for a customer. She has two interfaces. The party that refers to her might have determined which interface she has, but even though both parties are referring to the same person, she might do different things for them.

Objects may have the same ability. A type of object can have separate interfaces. Interfaces are groups of an object’s members. (The elements inside an object’s can also be called *members*.) Another object that refers to the object can decide 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 here.

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

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

You might not always start with the largest container. Maybe one symbol is highlighted. Then this symbol might be what the diagram is about and the focus might be mostly the aspects of the highlighted symbol. Then you might start at the highlighted symbol and look around. There is probably no one way of doing it.

Triangles are actually sub objects too. One of the differences with circles is that triangles basically melt together with their container.

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

### Conclusion

Hopefully this adequately introduces the basics of the Circle language.