|  |
| --- |
| Circle Language Spec: Misc Diagram Topics |

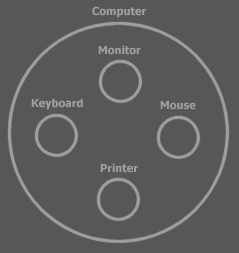
## Introduction

< Introduction of the old 2004 Symbol language documentation. >

This first chapter explains the basics of programming. It should introduce you to programming whether you’ve programmed before or not.

### Idea Subdivision

Contemporary programming is all about splitting up a large idea into smaller ideas.



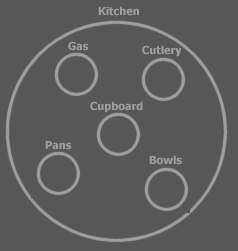
Here you see the idea of *the computer*, split up into four sub ideas: the monitor, keyboard, mouse and printer.

In Symbol, the main symbol for an idea is a *circle*:

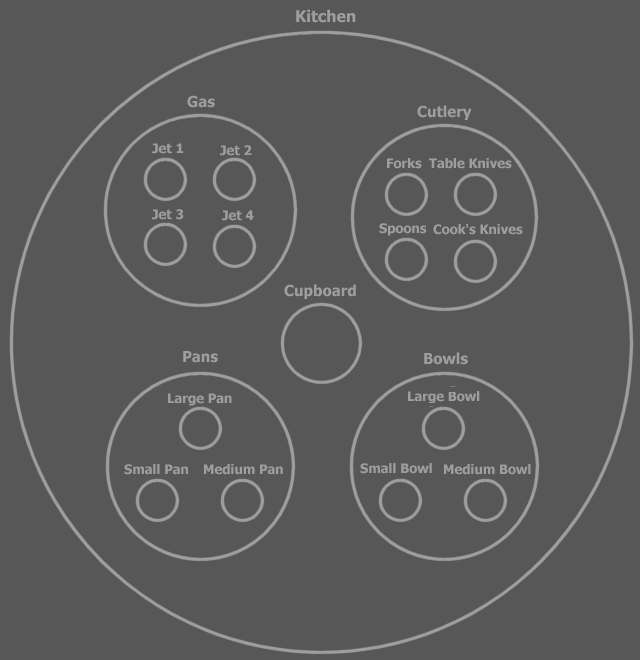


Each sub idea works independently of the other ideas. **Monitor** does its bit, **Keyboard** does its bit, etcetera. It’s the *super* idea that ties the sub ideas together. That means that the computer makes the links between monitor, keyboard, mouse and printer. The super idea combines the sub ideas and manages the communication between the sub ideas.

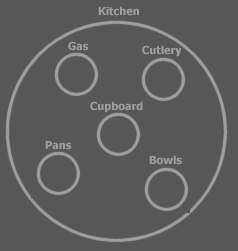
Ideas, so both super ideas and sub ideas, are called objects. Each object is responsible for its own bit of the system.



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 only need to see the top layers of the split up.



This makes object programming an excellent way to keep overview over the system as its complexity builds up. The understandability of the design is dependent on how outspoken and clear the split up into ideas is, especially the top layers. I think that the subdivision in objects is the most important thing in your software. So splitting up an idea into sub ideas, something that even non-programmers can do, is the most important thing about programming.

### Reusing Ideas

Well design in grouping ideas leads to the *reusability* of ideas. For instance: the idea of a *button* is very reusable. It’s used everywhere. If you program the perfect button, nobody ever needs to program a button again. You can 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**.

### Skill comes with Time

The more experience you gain in splitting up large ideas into smaller parts, the better your designs will get, because in time you’ll get a clearer view on how to make a system complete and how to subdivide systems. At first you may imagine that the computer will do a lot of the work automatically for you. You’ll think: “When I do this and that, the rest will be done automatically.”. However, if you build a system, you have to give *everything* a place yourself.

### Programming is Never Easy

Some texts explaining programming environments try to sell it to you that software programming is easy. That is never true. Those texts are written to *sell* the programming environment, not to tell the truth. Those texts claim that the programming environment will do most of the work *for* you, while in real life, what the programming environment does *for* you, is just run existing programs, not program new software. The text will say you’ve programmed something, while you’ve really just run something that was already programmed. Only if you want to take *no* decisions on how the software is going to work, programming is easy. But then it’s not really programming. If you want to make decisions about how you want your software to be, software programming is *not* easy. In programming, you have to give everything a place *yourself*. If you reuse someone else’s object, you have to be aware of what the object will do for you. Worse, many times you encounter problems with someone else’s object that you can’t fix, because you can’t change the inner workings of the object. This can lead to wanting to program everything yourself and if you’re stubborn enough, or fed up enough, you become *me,* and actually try to start to program everything yourself. Learn to work with other people’s objects. I’m working on that too. *And* I’m trying to make *the* **Button** for which no one wants a replacement. Or if they do want a replacement, they’ll still want to use the base of my button as a foundation.



### Many Don’t Get It

Many think it’s not important to have a good subdivision in sub ideas and to come up with good names. They think that something just needs to work and then it’s ok. Consider what happens when you take away good naming and good subdivision: all you’re left with is procedures calling eachother for no apparent reason until they resolve in machine instructions, the names of which are fortunately already picked by Intel(\*. The clarity of code is dependant on the clarity of names and subdivision. The less clear the names and subdivision are, the less sense the code will make.

Sometimes I spend most time thinking about a subdivision of code and about names, not testing if something is doing what I want it to do. I can spend ten minutes figuring out a name. If I don’t give it a good name, the code is not going to make sense.

An identifier is the only thing that can tell something about the contents of a symbol when its contained symbols aren’t shown. A kitchen consists of hundreds of things, but if you say ‘kitchen’ you really know enough. If it’s given a bad name, then you don’t know what it does. Then the only way you can know what it does is by looking at its contents and hope that good names are used there. If the names inside are no good either, there is still no way of telling what it does. You can’t tell what it does from just procedure structure and object structure. Eventually there has to be names. If all names suck, then you rely on interpreting machine instructions. If Intel(\* would have given indistinguishable names to machine instructions then you would be lost. You can’t see anything in the code then. All you could rely on is documentation saying what’s what, if such documentation even exists. In some cases where you use proper naming, you hardly need to supply any documentation.

Even when you don’t think a lot, when giving something a name, even then naming is important. The names, however quickly and poorly chosen, are distinguishable or the consequence is that you can’t see what your code does, unless you look into what’s under the name, hoping you’ll recognize something there.