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

## Overview

This documentation is an attempt to describe a programming language called Circle. It aims to be a diagram notation for visualizing computer code.

The documentation tries to explain how coding concepts might be expressed in this Circle language: concepts like objects, classes, relations, interfaces, events, commands and parameters.

The Circle language for now is a specification in writing. At the time of this writing, no finished program was made yet. Only experiments were done.

The documentation tries to shed light on the following topics:

### Overview

This overview, which attempts to summarize each topic.

*Introduction*

Tries to gently introduce this language.

*Basic Diagram Elements*



Tries to make it a bit more formal with an attempt to give a more or less complete overview over the basic elements that might be seen in the diagrams, like circles, triangles, crosses, squares and lines. This documentation describes what they might mean in the diagrams. Details might follow in later text.

### Objects



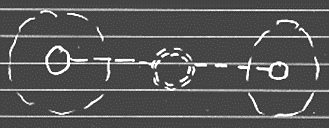
This documentation aims to describe what objects might be in computer technology and basic concepts about objects. It also tries to shed light on drawing out objects using the Circle diagram notation.

### Classes



This documentation attempts to demonstrate the concept of *classes*. A class is sort of like a special object, that aims to describe the characteristics and behavior of other objects. The other objects might select that special object as their class and the idea is that their behavior is then guided by the rules described by that class. The basic ideas may be explained, but the concept of *static* is intended to be worked at a later time.

### Relations



When an object connects to another object, it is like a relation between the two is established. This documentation tries to clarify the concept of relations. Even though objects might relate to an arbitrary set of other objects, the concept might also be about how *classes* set the guidelines for the kinds of relations objects might have. Relations between classes form a sort of *model* of relations for objects.

### System Objects



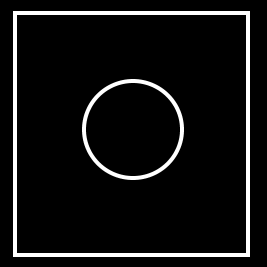
One possible application of how this language might run in practice, is one where systems may be composed of *system objects*. System objects could be the *actual* objects, that a system runs on. Maybe this is a specific use case of the language, but describing it, seems to have allowed more generally usable things fall out. One of the intentions of system objects could be, to manage *relations* between objects. They might also control *system aspects*, such as **Class**, **Interface**, **Execution** and **Data**. Those aspects could be controlled through *system commands*. The story moves on to assignment commands. The idea is that assignment command copy an aspect from one object to another. The *System Objects* documentation alsotries to show connectors, connections and different notational forms to regulate these aspects.

### Commands



Commands could be seen as executable objects. Other synonyms for commands might be *methods*, *procedures*, *routines*, *subs* and *functions*. But in the Circle Language Spec those executable objects would be called *commands*. In some programming languages, commands might not be objects, but constructs, that are not exchangeable with objects. But in Circle an attempt is made to see them as objects, that happen to be executable. Commands do seem to have specific behavior. This documentation tries to evaluate this behavior.

### Parameters



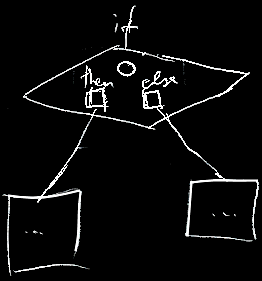
Parameters are like instructions passed along with a command that could make the command behave differently. This documentation tries to view parameters as a relationshipbetween commands and objects. It entertains an idea of how *command parameters* and *class methods* might be interchangeable, as if they might be one and the same concept. It is not sure, if this idea will still be in there in the future. Also a diagram notation of commands and parameters is suggested.

### Modules



A module would be like a little world in which objects and commands live. A module might be a site, a computer program, a library, a name space or other kind of module. A module may group together functionality or data. A module might be thought of as an object with a special property: something that might be called *global* access to its sub-objects. This is another suggested description of that property: Anything directly inside the module can be directly referenced from anywhere within that module.

### Execution Flow



The idea of execution flow statements is that they may regulate the order in which things are executed. Several execution flow statements are proposed. The key examples might be **If** and **For** statements. An attempt is made to build up a complete set of execution flow statements, to explore what could be desired from the diagram notation for it.

### Black Boxes



It could be said, that *Black Boxes* have to do with *Public and Private*. *Private* might make something only accessible inside an object. *Public* couldmake something accessible outside an object as well. That way, the inner workings of an object may be hidden away and this might be a way to work with the input and output only.

The notation might be changed compared to what is described here. The current version tends to focus around something, that could be called the *friend* notation. This might be because the current design of the notation seems to be made with a bit of a different view on objects and commands. Commands were sort of considered more independent and not necessarily part of an object. Which commands might access the private contents of which objects might be less obvious, when a command would not necessarily be part of any particular object. The effect seemed to be, that commands might not get access to the private contents of an object, unless that command may be declared a *friend* bythe object. Friend declaration might create somewhat of a tighter bond between the command and the object. Thinking of commands as object, seems to have affected the direction into which the design black boxing went.

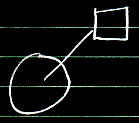
In the future, a different black box method might be suggested: more like it is in other programming languages. The friend notation might still be preserved for particular use-cases.

### Interfaces



This chapter attempts to describe the concept of interfaces along with their diagram notation. Up till now the idea of interfaces was sometimes left out of the stories. This chapter tries to catch up with that. There are also attemps to describe: interface assignment, interface reference and relations that might form through the use of interfaces. As interfaces might be applied to *objects*, interfaces might also be applied to *commands*.

### Events



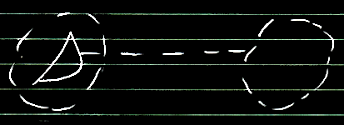
Events might be thought of as notification calls. Call receivers might subscribe to an event. The call sender might send a call to the receivers at times. Another variation of the events concept, that the world seems to use, if not mistaken is the *‘the observer pattern’*. The term observer may help here, to realize what an event is for: watching out for something to take place, to possibly take action. The sender might decide what the message looks like. The receiver might supply a command, that should have an interface, given by the sender.

The concept of events might be seen as a combination of constructs: an array of command references, a command interface, then supporting the command interface, registering the command inside the array of command references and then calling the commands in the array at times.

Even though It might be established by a combination of other constructs, it seems to be of enough use, that Circle attempts to accommodate it as a separate construct, like some other programming languages seem to do.

The documentation also tries to show, how events look in Circle's diagram notation.

### Inheritance



This part of the documentation attempts to describe the concept of inheritance inside the Circle language. This concept seems to have been part of some object oriented programming languages for a while. In certain languages one class might inherit from another class, to take over its characteristics before the derived class might get extended or altered. When the notation in Circle was worked out, different variations on that seemed to fall out, that might not necessarily be present in all object oriented languages. Next to *class inheritance* (could be thought of as the *traditional* form of inheritance), there could also be *object inheritance*. This might mean, that one object takes over the characteristics of a specific other object. The other object seems to melt together with the new object. When maybe changing base members of the new object, the base object’s data might also change. This could be called *object inheritance*. More forms of inheritance may be proposed. A collection might be inherited from, so that an existing 'table' of items might be extended with 'new columns'. This could be called *list inheritance*. There is also the idea for a construct, where a *system* or modulecan inherit from another system, taking over its characteristics, and the derived system can be altered, augmented, specialized and extended. This might be called *system inheritance* or *module inheritance*. If the original system would change, this may also change the derived systems. (Other technology might establish this with what may be called 'proxy classes' and extending auto-generated proxy-classes with additional aspects.) So the idea is that things can be inherited from to render specialized versions of a class, object, collection or module.

Such specialization might also be applied to *commands*. The inheritancedocumentation also aims to describe otherconcepts in the area of *specialization* and *extension*. Inheritance might be thought of related to the concept of *interfaces.*

That was the more rounded updocumentation. Next an attempt is made to list the documents, that might not be as well polished. The descriptions below, however, might be a rounded reflection on the topics.

### Type Control

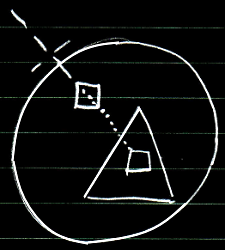


Objects might relate to free to choose sets of other objects. But when the class of an object is set, some relations might be determined by the class. This could be considered a form of *type control*. Another form of type control might be assigning an interface to an object reference. This may change the types of objects that might be assigned to the object reference.

Binding an object like that could be called *explicit* declaration. Not binding an object or object reference like that might be called *generic* declaration. Both concepts could be seen as *type control*.

This documentation folder might have one document, with ideas that are possibly not as well polished, including an original description worked out in a previous version of the design of this diagram notation language.

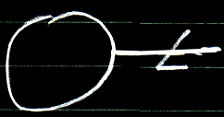
### Object Resolution



Object resolution might take place when a reference to one object is more or less obscured by another object, shadowing or overriding the original object. (It also might apply to commands.) A reference might be set to an object, but instead the reference might be sort of non-consentiously pulled towards another object. There may be multiple situations in which object resolution takes place, like: overriding, overloading, shadowing, ambiguity, implicit conversions and default members. At first it was thought, that these kinds of situations might be due to ambiguity and the textual nature of code, but in the Circle diagram notation when drawing out the situation in a diagram, it seems to fall into place, that they might be more like automatically detoured references.

This documentation might be less smooth. It may contain ideas but partly described, not as refined into polished up text.

### Conditions



An object might set rules for what kind of values may be filled in into it and a command might be given conditions on whether the procedure might be started or not. This might perhaps be done by adding code to a procedure (or for an object, to the **Set** command). From that code may follow, that depending on certain conditions, the execution of the rest of the procedure is or is not run. That just may have described a low-level way to handle conditions. But conditions could be centric enough in programming, that they might become a construct in the Circle diagram language. A (value) object or command might propose a set of conditions, to evaluate whether the value may be assigned or if the command is allowed to execute. The documentation may be not so rich yet, apart from some loose ideas.

*Object Order*

By default, objects inside a collection might be put in an arbitrary order. The *Object Order* concept may allow sorting a list, or putting things in a manually chosen order, and also perhaps a diagram notation for it. This documentation is a collection of ideas, that might be polished up to be made neater.

### Pointers



The intention could be to move pointer-to-pointer related issues from being in more than one place to this chapter. This may or may not have been done yet.

*Misc Diagram Topics*

It would be nice if diagrams could be automatically drawn out: if the metrics and positions of the shapes and lines could be automatically done. Suggestions could be found this documentation about aspects of positioning, that might make diagrams look tidier. Also described might be ideas about how colors may be used to express things. This documentation might still be loose ideas from previous versions of descriptions of this Circle programming language.

### Summaries of Other Languages

Attempts to summarize other languages, mainly C and C++, in the form of bulleted lists. They were made around the year 2004 during an analysis of the grammar of C and C++ to compare it to this Circle programming language.

*Previous Version*

The previous version of a description of this programming language.