# Circle Language Spec Strategy

*Author: JJ van Zon*

*Location: Oosterhout, The Netherlands*

*Date: February 2008 – April 2020*

## Contents

Contents 1

Introduction 1

Earlier Goal 1

Later Goal 2

Theme Picking 3

Focus on Easy Themes 3

Redo Easy Themes 3

Rules 4

Plan the Specifics 4

Project Order 4

Emphasized Again 4

Productive Writing 4

Writing Efficiently 5

Standard Work in Next Projects 8

Conceptual Explanations vs Diagram Explanations 9

Do not cover uses 9

## Introduction

This document describes the strategy and goals of the project *Circle Language Spec*.

*Circle Language* is an idea for a new computer programming language, mostly about a diagram notation that shows things, that go on inside a computer. It could become a way to navigate and write code.

This document explains a lot of details about how the work was done. The project was split up into multiple sub-projects.

The document mostly talks about strategy of work already done. That does not mean, that it might not have some value later.

## Earlier Goal

A while ago the goal of the project was to describe the ideas. It was meant to preserve the thoughts about this language. This was done, by making a specification of the new programming language, adequate as a starting point for turning it into a usable product. Then the idea could either be sold, given away, implemented or not worked on for years and then the ideas would not fade away. Theoretically if anyone wanted to, they should be able to continue working on it from that point on. That would give peace of mind and it would open up possibilities for the idea.

Later in the project, that goal was mostly realized.

## Later Goal

Later the goals for the new programming language went beyond describing the existing ideas adequately.

The goals of the sub-projects would no longer be just to describe each element of the system.

The goals of the sub-projects would lean more towards turning the new programming language into a finished and usable product.

Product is a loose term here. It can mean a working piece of software in which you can use the new programming language. But a product could also be a specification on paper of what the new programming language would look like.

Some stuff got finished.

Some things were put off till later.

New things were imported into the project.

Priorities got changed.

Goals got redefined.

There could be multiple intermediate goals. Each goal would be about taking the language to the next level.

- Documentation: Adequate OO Paradigm (mostly finished)

- Documentation: Automatic Diagram Organization (part finished)

- Programming: Program a version (part done)

- Business: Release

- Documentation: Added Paradigm: Concepts

- Documentation: Advanced Input/Output

These phases are bigsteps. Each of them is a big step.

The 'release' thing, is most important now (2020-04), and the rest of it is not.

The first phase is 'Documentation: Adequate OO Paradigm'. It only consists of a couple of more sub-projects. It is mostly done. The end result would be pretty much a full specification of an object oriented diagram language in which object oriented systems can be expressed in a large amount of detail.

The second phase is 'Documentation: Automatic Diagram Organization' is only part done. The thing is: when diagrams do not organize themselves automatically (for instance positioning of the shapes and curving of the lines), the effectivity of the language might be decreased, possibly to a point that the intentions with the language are not reached. If the way diagrams organize themselves is automatically worked out, the usability of the language might very much increase.

Programming: This was the point at which, it was found a good plan to actually program a version. Even though a lot of the language would not have been designed yet, some experimental things have been programmed. One fear was: making something, that would later be thrown away, would the language change completely. But the base of the language was not expected to change drastically. Another fear was to lose time programming, that was better spent on the language design. The hope was, that a working version would tell where the big problems are would indicate solutions to problems initially not possible to get your head around.

At that point it might actually be released it to the public. The goal used to be to patent it, but that is no longer the case. Open sourcing is the goal now. Not sure how to make money off of this, and it is no use just sitting on the idea having time just pass by.

That is where it's at right now. The other steps in the list above are possible future steps, but possibly not necessary.

## Theme Picking

This section covers strategies for rough planning and *theme picking*. The most important thing may be to focus on the easiest stuff. This is not just lazy. It has some advantages.

### Focus on Easy Themes

Less finished-up documentation was moved to the bottom of the documentation folders. Only if a topic is better finished up, it might be moved back to the top of the documentation again.

The idea is to first describe things, that are already clear ideas. First skip the harder themes, with less of a clear view. There are many themes, that can be relatively easily worked out. It is quicker and the work is more 'overdue'.

One reason for this is, to get more work done quicker. Another reason is, that the current projects are about making *existing* ideas easier to pick up by others, instead of covering *new* ideas. Another argument is: if you document harder topics, you will create documentation of lesser quality, less accessible. So the right decision seems to be, to do easier topics first.

### Redo Easy Themes

Some of the more recently done documentation is still tough to read.

This is, because when writing the documentation, the concept was still being worked out. The documentation was written in a way easy to write. Sometimes it was tough enough getting a concept straight on paper. You can not blame anyone or anything for the material first to be written in a way in, which it is easier to *write*. But later, the material and the reading order might be changed, so it is easier to *read*.

Hazards that might make the reader loose confidence in you:

- Too much

- Too difficult

- Too much junk in it

- Not finished = not worth reading

But maybe this is a bit harsh.

### Rules

- Focus on themes easy to work out is a pretty much strict rule.

- Simplify existing chapters.

- Each theme: only a couple of weeks, is the aim.

- Use *themes* as smallest unit for time planning and progress monitoring.

- No separate progress monitoring for the smaller parts.

- Try to quickly make documenting existing ideas a past stage.

- Documenting existing ideas = most important

- Do not plan for the difficult topics; the difficult topics might become easier in the future.

### Plan the Specifics

- Which easy subjects *exactly?*

- Which existing topics to simplify *exactly?*

- Which tough subjects *exactly?*

- Define all projects

- Make rough planning

- 8 easy subjects: 2 months

- Simplifying existing topics: 1 month

- Tougher subjects: how long?

Plan for easy topics only. The idea is: you do not know how difficult the harder topics will be in the future.

### Project Order

This paragraph may be short, but this is quite important. This is the planned order in which to do things:

**- Easy subjects**

**- Simplify existing subjects**

**- Tougher subjects**

(perhaps by the time you get to them, they will be easier)

### Emphasized Again

The idea is to work out the ideas that are already clear, so that the existing ideas are easier to pick up by others, instead of working out concepts, that are not crystalized out very well yet. This should be quicker and easier.

## Productive Writing

In contrast to the previous section, this section is not about rough theme picking, but more about strategies for when you have picked a theme and want to start working on that theme.

This section covers mostly strategies about the details of writing, but also some other project execution strategies.

### Writing Efficiently

There used to be too many demands on the articles to produce.

It was a big project and it was going sort of slow.

A lot of extra effort went into make the articles 'perfect'.

But then it was decided, to focus more on making the articles *adequate*.

A thing to get used to might be, to have them look less 'perfect' from now on.

There can be several kinds of tasks involved in projects for writing these articles:

- Collect existing ideas

- Brainstorm about systematics

- Organize & reformulate brainstorm texts

- Determine article list

- Converting brainstorm texts to articles

- Start over

- Use old content as a sort of cross out list.

- Write articles

- Brainstorm about different approach

- Adapt articles to different approach

- Add references to other articles

- Process details

- Reading over

- Folder organization

- Scatter texts from old documents across a new form of documentation subdivision

The project might have been executed in a too perfectionistic way.

The goal of the language spec could be kept in mind more clearly.

You may want to ignore the side-issues.

This general goal could be repeated at the start of each sub-project.

Maybe side-goals cannot be in there just for the fun of it, because each extra hour seems one hour too much, when things are going slowly.

- Creative vs. productive:

- Form a clear idea in your head.

- It should be clear in your head,  
before you can make it clear on paper.

- Your whole consciousness may need to understand the concept first.

- You are not a production machine.

- You may not be able to make this, if you only focus on producing articles and do not even understand the concept anymore.

- When you have totally forgotten the workings of an important concept, written about before, then maybe the focus has been too much on producing documentation, rather than forming an idea in your head.

Below, each task is commented on.

The focus lies on article content, not on perfection.

- Collect existing ideas

You might give some importance to collecting all existing ideas (notes, 'idea boxes') at the beginning of a project. That does not change, even though it might be quiet some work. The collection of those ideas may actually be a good starting point, for the eventual content. You might miss out on something good, if you do not do it.

- Organize & reformulate brainstorm texts

The method employed for organizing and reformulating brainstorm texts seems fine, but sometimes it is better to just derive a topic list from the idea texts and start over.

- Creative thinking

Brainstorming about systematics might not ever become easier. You might not want to see this as production work. You may want to do creative thinking and take some time for it to become crystal clear.

It may be a good idea, to not just stick to the frame of the project. It is *one* programming language. You may want to keep the bigger picture in mind, even though a project is to work out everthing about a specific subject, for instance: *interfaces*. It should be a good idea to look at the broader view too.

- Determine article list

Sometimes there was too much focus on explaining each concept in its own separate article.

That might be too *form-*oriented.

It may have happened in an effort, to turn the work into countable reliable items.

It might have been realized too little, that it should be about *creative* thinking much of the time.

Work may have been regarded production work too much, or to look for the easiest way to get the set of articles done.

It is probably better to focus more on *creative* thinking, instead of *productive* thinking.

Maybe split up the material into less articles, if that makes it easier to finish the project.

Make it a single article, if you must.

Sometimes, when things are too complicated, one might ‘artificially’ split up the story into articles. But that is only done, so to get a clearer view on things, so then it might be alright.

Sometimes too much effort went into making the article list a set of concrete tangible concepts. Sometimes when there were rules, that apply to multiple concepts, the rules were repeated over and over again in multiple articles. It should be allowed to isolate a rule into a separate article and to not repeat it everywhere.

- Converting brainstorms to articles

You might want to be more satisfied with having a set of stories, the way they were in the brainstorm texts. Do not give the articles an entirely different subdivision, than the brainstorm texts have.

- Article content

**defend**:

Try not to *defend* the system too much.

**compare**:

Try not to *compare* the system too much to other systems.

**how & why**:

Try to avoid talking about too many because’s and *how* and *why*.

**blunt**:

Tend to be more *blunt* about how things work.

**level or knowledge of the reader**:

Try not to worry too much about the *level or knowledge of the reader*.

The material does not have to be read by just *any* person or by 'the average programmer'.

It does not have to explain the workings of the CPU, the working of other languages, etcetera.

If, when writing, you experience confusion about something, you may need to explain it, for yourself.

G*eneral* notational rules might be covered too much in *specific* diagram articles.

The notational rules could be covered separately.

They do not need to be referred to necessarily.

Try not to worry too much about the implications of *automatic containment*.

**exceptional cases**:

Try not to worry about *exceptional cases*.

**implications**:

Try not to get paranoid about not considering the *implications* of things enough.

**uses**:

Try to avoid explaining all the uses of the language.

It would be hard to cover them all.

The basic building blocks should be provided, not necessarily examples of what to do with it.

**delete**:

Try not to be afraid to permanently delete texts, that fall under any of the categories above.

You migt get paranoid about the details, but they might come to light during an *implementation* of the system.

The danger of too much detail, might be to impose too many rules, that you later must break.

Or that a lot of thoughts go into rules, that later cannot be upheld.

The list above may work well as a set of rules for a sort of ‘reach goal’ / 'prototyping' part of a project, as opposed to a ‘make perfect’ / 'finishing touches' part of a project. But perhaps you should not cover details at all, because they also might burden the reader.

- Start over

Sometimes when you have a big collection of ideas and brainstorms, it may work better, to just extract a topic list out of it and start over.

- Cross out

Later you can use older material as a cross-out list, to delete older stuff already covered by a newer story and delete stuff that is no longer interesting and to extract stuff that might still be interesting.

- Adapt articles to different approach

Sometimes not enough effort may have been put into this.

Adapting articles to a different approach is quite some work.

You might see this as production work, but you might not want to see it that way.

Try not to focus on *getting it over with*.

Try to explain the new *concept* well and getting the details right in your head.  
Sometimes facts were left in, that just might not be right, just to get things over with.

That may leave the story in a not so good state.

- Article form

Too many hours might go into adding references to other articles everywhere.

Too much time might go into worrying, whether everything referred to, has already been explained or not.

Too much time might go into giving the article a perfect form.

- Processing details

At the end of the project there might be a list of details left. You first might want to consider, not to adapt the articles to those details at all.

- Reading over

Consider to not read over the articles as much.

At the end of a project, consider not to read over all articles. It might take a lot of time.

- Folder organization

- Simplification in folder organization seems good.

- But maybe do not worry too much about ‘it’s easier for the reader this way’ and stuff. Now might not be the time, to be friendly to the reader, given you have so much documentation to produce.

The story above, might actually also try to break things up into logical units too much. That might be too form-oriented. Sometimes the rules seem to contradict. They're just guidelines.

### Standard Work in Next Projects

In each next project you pick, however time-costly, *do* consider processing the idea box / loose ideas you might have. Reformulation of loose ideas, might actually result in a readable, structured article covering the topic. That is one of the strategies, that might actually lead to good text.

Each project, in which you update documentation, could be trailed by an update to the redirection pages, that tell something about that documentation.

### Conceptual Explanations vs Diagram Explanations

Conceptual explanation and the demonstration of the diagram notation were separated in the past. This was done, because sometimes you can conceptually explain something without encountering any problems, and the problem may only be how to express it in a diagram. In that case it can be easier to first draw out a conceptual point of view, and that makes it easier to work out a diagram notation. The conceptual point of view then does not change, but an imperfect diagram notation may be changed later, when your vision on it becomes clearer. Also it allows you to put everything into the context of one single language, even when not every concept has its own diagram notation yet. Also: sometimes when you work out the conceptual explanation first, it becomes easier to figure out a proper diagram notation for it. However, you might later want to merge conceptual explanation with diagram notation explanation in certain cases, because this is often easier on the reader.

### Do not cover uses

There used to be the idea, that for the interfaces theme of the new programming language, it would be good to read more about design patterns. But reading about design patterns at this stage, might not actually be a good idea for progress.

The reason why considering design patterns could be harmful is: if you do it for the *Interfaces* articles, you would read it to orient yourself in the different uses of interfaces, in order to explain those in that documentation section. But one of the 'rules' was not to cover all the different uses of a concept, but just the blunt description of the concept. Sometimes *uses* add necessary explanations to a documentation section, but the rule is to not cover all the different uses. The implementation of interfaces in the new language can be bluntly explained, and it should not raise much discussion among programmers, if they already know how interfaces work in other languages.

Also: design patterns might introduce new concepts, that might inspire to revise large parts of the language. That might be harmful to progress. It might be better to work out existing ideas the way they are now, instead of changing them all again and having no straight forward story to tell.

So those things might make reading about design patterns at this stage not such a good idea.