# Circle Language | Construct Drafts | Old Ideas

## Development Notes

### Symbol Development Notes

Stepping through a procedure without seeing that you enter a whole tree out of procedures, should be a standard stepping mode.

## Contents

Contents

Introduction

## Introduction

J Code is the compiler of the Sigma operating system. It converts one type of code to another.

There is a standard way to describe a programmatic system: the system document, which is a system specification document. You specify a system object orientedly.

The main language of Sigma can be said to be Symbol code. Symbol code is a diagrammatic representation of the system specification document. When speaking of Symbol code I may be speaking of the diagrams, but I can also be speaking of a system document.

Symbol code can be compiled to Sigma module machine code.

J Code and J Core are basically the same project.

This project involves the following:

* J Core
  + All of J Core
  + Sigma module
* Language Definition
  + Symbol language
  + Symbol document
* Translation
  + Symbol to Sigma
  + Parsing
  + Symbol to Text
  + Text to Symbol
  + Basic to Symbol
  + C++ to Symbol
  + Symbol to Basic
  + Symbol to C++
  + ***Optimization***
    - Procedural optimization
    - Machine code optimization
* Symbol View and Interaction

Each subject has its own document. This document only describes the subjects in general.

## Symbol Database

Symbol language is stored in a J Database.

### Old

### System Specification

The symbol language includes a system specification document class. This class will be used to contain the source code. The document is read to be able to produce the Sigma module.

### Document

The document is encapsulated by a single class that represents the part of the software through which the Symbol code data is managed and in which it is stored and guarded. Calling the members of its interface will not harm the integrity of the Symbol diagram.

The document returns the diagram data, provides edit methods, manages selections, allows undo, etc. Almost all possibilities for the user of the Symbol Enviroment have a place in the symbol document.

#### Copy and Paste

(and Cut, Save Selected and Merge)

Cutting is the same as Copying and then Deleting. Saving Selected will work the same as Copying. Merging will work the same as Pasting. This narrows it down to Copy and Paste. I will explaining Copy and Paste in detail.

No diagram will have classes, references or parents that do not exist in the same diagram. If a fraction of a diagram is taken when copying and classes of objects in the fraction are not in the fraction then dummy classes will be added and taken in use. The same goes for parents and references.

An effect of that is that cutting and then immediately pasting does not keep the situation in tact. Classes that weren't cut are no longer referred to by the cutted and pasted symbols, but they will have a dummy as a class.

Only the outer parent will not be dummied. There will be –1 parents in the fraction, which will become the parent the fraction is pasted in.

Just about the only difference between Copy/Paste and SaveSelected/Merge is that SaveSelected/Merge has to use a whole SymDoc as a 'clipboard', while Copy/Paste can do with only a SymDocTable which excludes indexes, selections and many other things.

##### Paste

Usually paste comes after copy, but it's easier to understand in this case. The clipboard contains a valid diagram. When pasting you first add to the main diagram all the symbols in the clipboard with all their properties except parent, reference and class. Mind that adding a symbol doesn't mean that the new symbol is at the end of the line. the new symbol could be anywhere in the table. While adding all those symbols from the clipboard to the main diagram you build an array that contains the positions of all the newly added symbols. This same array serves as a conversion from the clipboard diagram symbol positions to the new main diagram symbol positions.

( all the classes, referees and parents are there now safe to be referred to if you convert the clipboard positions to main diagram positions )

You know *which* of the diagram symbols were just added because we built up an array saying which ones were added. Then you change the parent, reference and class properties of the added symbols not to what's in the clipboard, but to what's in the clipboard converted to positions in the main diagram.

##### Copy

In every SymDocTable there is an extra field 'Vector' that serves

as a vector table.

A secondary SymDoc or a SymDocMain will function as a clipboard. You add as many symbols to the clipboard as should be copied and set all properties except Parent, Reference and Class. As you do that you fill in the vector table in the main digram: diagram position -> clipboard position which means that you don't change ALL the entries in the vector table.

After all symbols have been added to the clipboard you will change the parents, references and classes in the clipboard to the value in the primary doc converted with the vector table so they become references to positions in the clipboard and not position in the diagram. If a class wasn't copied then the vector table indicates -1 for it and then an extra symbol is to be added to the clipboard representing a dummy class and an the entry in the vector table that converts the not copied class will convert it to the dummy class. Positions of these vectors to dummy classes are kept in an array. Then the process of changing parents, references and classes can continue. Ofcourse when I talk about a class the same goes for parents and references. After copying, the vector table should be reset. All the selected symbols' Vector properties are to be set to -1.

We kept an array of positions of vectors to dummy classes these vectors should be reset to -1 also.

The residence of the vector table is best in the main symbol table for it would be costly to create an array with 60,000 records every time you copy any small bit of a 60,000 symbol diagram.

#### The below information about Document is obsolete

In this section I will describe abstractly three things about the document: what is explicitly stored (“Stored”), what is read only and what is random access (“Properties”) and the methods of the document class (“Methods”).

Keywords regarding the document are: integrity, indexes and abundance.

#### Stored

The document explicitly stores:

Main data:

* Symbols
* Name()
* Public()
* Type()
* In()
* To()
* Of()

Symbols is the number of symbols.

Name contains the names of these symbols.

Public contains whether a symbol is public or not.

Type contains the type of which a symbol is. That can be a circle, a triangle, a square or invalid.

‘In’ contains for each symbol a pointer to the symbol the symbol is inside. Thus this is the main data of the encapsulation structure of a system.

‘To’ contains for each symbol a pointer to the symbol this symbol refers to. Thus this is the main data for the delegance/reference/call structure of a system.

‘Of ‘contains for each symbol a pointer to the symbol that is the descendant class of this symbol. Thus this is the main data of the inheritance structure of a system.

Indexes:

* ValidSymbols
* ValidSymbol()
* SymbolsInMe()
* InMe(,)
* SymbolsToMe()
* ToMe(,)
* SymbolsOfMe()
* OfMe(,)

ValidSymbols is the number of valid symbols.

ValidSymbol is an enumeration of pointers to all valid symbols.

SymbolsInMe is the number of contained symbols for each symbol.

InMe is an enumeration of pointers pointing to all contained symbols.

SymbolsToMe is for each symbol the number of symbols refering to that symbol.

ToMe is an enumeration of pointers pointing to all symbols refering to the a symbol.

SymbolsOfMe is the number of decendants for each symbol.

OfMe is an enumeration of the ancestors of a symbol.

Cursors:

* SymbolCursor
* SymbolInMeCursor()
* SymbolToMeCursor()
* SymbolOfMeCursor()

SymbolCursor returns the symbol currently worked with. That means which symbol in the main data we are taking actions upon.

SymbolInMeCursor returns the position we’re currently at in the enumerion of contained symbols of the symbol at SymbolCursor.

SymbolToMeCursor returns the position we’re currently at in the enumerion of symbols that refer to the symbol at SymbolCursor.

SymbolOfMeCursor returns the position we’re currently at in the enumerion of the ancestors of the symbol at SymbolCursor.

#### Properties

All of the members mentioned in “Stored” are properties. Some of the properties though are read only outside current scope, others are random access and others need to be altered using methods. That is what is explained in this section.

#### Methods

There are methods that can return to you object relational information and there are the main methods with which to manipulate the diagram. The object relation information simply derive their return values from the stored data. The main data manipulation methods are the guards of the diagram data. They put restrictions on data manipulation that guard the data’s integrity.

### Brainstorm

#### System Specification

The symbol language includes a system specification document class. This class will be used to contain the source code. The document is read to be able to produce the Sigma module.

## Translation

### Optimizations

#### Boolean Algebraic

Everything should eventually lead to storage.

What is storage?

If you take storage out of the code then do you have like one large boolean expression?

I don’t know.

It just worries me that If is a boolean operator

If x = 43 then 3

#### Procedural

##### Inline

Private procedures can be compiled inline except when infinite recursion is possible.

You can also choose to compile all procedures inline. This means that when you have compiled the module you don’t have separate procedures anymore, so you can’t have separate interface members, so no interfaces and no classes. That way the whole object oriented organization collapses and you have one pile of spaghetti code. Spaghetti code that is *way* faster than the organized code. However, you can’t use the classes of the module anymore. You’d have to have the module’s source code to be able to program with it again.

##### Inline Module

A module can work faster if it doesn’t supply any procedures. The only call to the module then is to simply run it. All procedures inside the module are compiled inline then. However, it must be possible for the module to create its own memory objects, even though there aren’t any separate classes anymore. The inline module doesn’t have vector tables. It does have a table with the addresses of memory object specifications. On top of that it doesn’t have procedures, but instead one pile of spaghetti code.

##### Inline Module Usage

If you choose to compile a module inline then you have a faster module. This code cannot be called by other modules though. The only way to use the module in another module is to have the source code and compile it into it inline.

However, the inline module can call another outline module’s procedures. You can’t call an inline module’s procedure, but an inline module can call an outline module’s procedures.

-----

Think over instantiating procedures when calling one. Consider recursive calls.... they do not say that instances of procedures are made and that there are classes and instances of procedures. It however needs to be so...

-----

Arguments can be passed through the stack, but sometimes you can use registers for it. Registers are way faster.

Randel Hyde is talking about making things faster. He speaks of being any good at Assembly partly is about making code as fast as possible. And yet he admits that HLA has been written prioritizing clarity SO that he seems abolish a technique that makes code faster.

So that seems to contradict.

you explicitly state the order of the instructions in code then if this statement of order is omitted, the assembly generator can choose its optimal combination of logical operation variations.

-----

For assembly generating software you could make tables drawing out logically equivalent operations and combinations of operations. some equivalents may generally work faster than others. The notation of an equivalent that is slower can be replaced by the equivalent that is faster by the assembly generator.

I talked about sequences of operations that perform a single logical operation.

I will call a squence of operation that perform a single logical operation a logical operation. The multitude of assembly equivalents I will call logical operation variations.

-----

Two logical operations may have their GENERALLY fastest way, but combining the two fastest operations may not be the fastest combination. Other combinations of equivalents of perhaps generally slower equivalents may together form a faster operation.

-----

Also: the order of logical operations may effect the speed. If you explicitly state the order of the instructions in code then if this statement of order is omitted, the assembly generator can choose its optimal combination of logical operation variations.

-----

The determination of the optimal combination of logical operation variations is done sequentially. The assembly generator looks at the sequences of logical operations. Which variation is appropriate is determined by its physical upstairs and downstairs neighbors. In case of jumps, conditional jumps and probable conditional jumps the upstairs neighbors are sometimes somewhere else than physically upstairs and sometimes they are.

Also, the flushing of the prefetch queue affects the performance. This does not mean that for comparing optimal neighbors it does not matter what the destination of a jump statement is. There are parts of the computer system other that the CPU prefetch queue that can be optimally combined as well.

Another optimization is keeping things close together in memory.

Another optimization is inlining: avoiding jumps.

I suspect that mr. Randall Hyde thinks of a computer software system as something so arbitrary that it can only be optimized by hand. That he thinks of squences of instructions including jump statements.

Sometimes, when input varies arbitrarily (within certain limits), you can't know exactly what will happen in the code in a single case. So there can not be an exact determination of what will happen in case of usage of arbitrary input. There can however be a determination of the ODDS of what will happen. The possible things that will happen is large. But these possibilities are all embedded into the code.

When you have a sequence of instructions that take an arbitrary value and limit it between -1 and 1 then this POTENTIALLY limits the possible progressions of the code.

Optimizing weighing out the odds of any possibility against each other is possible, but sometimes not enough. Of the possible things a user can do there are things that users are most likely to do and things that they are less likely to do. This requires arbitrary odds setting by the programmer. The programmer can add to the exact odds a logical odds facter which the assembly generator will take into consideration.

-----

This way, apart from anything the user can do to be most optimal as a whole, some things the user can do are more optimal than other things a user can do so that the things that the user will do most frequently are optimized the most.

-----

NOT SUMMARIZED:

When I say logical I don't mean Boolean.

There are tables for optimal assembly operation combinations. Those tables are different for different CPU's and different for any combination of computer modules. There are tables with assembly synonyms for logical operations.

-----

The table with optimal assembly operation combinations determines the fastest logical operation assembly synonym. It also determines the fastest logical operation combination. This is not tying together the fastest logical operations, because the combination of logical operations affects the speed so its the combination that determines the optimal form.

You can mess around with the order of the logical operations if a different order is faster. The programmer will say when the order of logical operations matters or not.

Jumps in code flush the prefetch queue.

Jump statements possibly vary the squences of logical operations arbitrarily. If no arbitrary input values are used then the program will still operate in a single way. Jumps can be omitted by tying the program up in one sequence without jumps, because the program will execute in but one way.

-----

When there is arbitrary (user) input than jumps can become conditional for real. You can't determine how a program will run.

You can however determine the possible ways the program can run. You can make an organization of the eact odds of the possible program runs. Only the squences that CAN execute deserve optimization.

More on it later.

-----

(PUT NEXT ICQ MESSAGE ABOVE)

Compute the optimization tables directly from the data you base these tables from, so that new computer configurations can instantly get an optimized compilation using the configuration data.

Substituting Intel mnemonics by readable names is a good idea. Being able to use the '=' sign in place of a comma would also be a great help.

Unreal forms of instructions, forms that don't exist on the 8086, but do have a logical meaning, can have synonyms that will work. Being able to use them will help.

I think that Assembly does not deserve to be used by hand. I think I should write a low level (but higher than assembly) alternative language the compiler of which looks up the fastest way to do it in assembly, so that all the damn shitty rules of assembly are hidden into the compiler and you shouldn't worry about how to optimize, because the compiler does it for you.

Gather up what useful things can be done with assembly instructions and use statements of useful things and let the compiler do its job.

Stupid synonyms, ones stupidly long, should also be translated to the optimal equivalient, maybe even especially stupid synonyms need to do this.

Perhaps there is a way to derive the synonym table from the behavior of every instruction.

A synonym can also be a synonym if they have different effects. Let's say the synonyms are equivalent except that one flag is undefined where in the other synonym it is set. If you won't use this flag you can replace the code by the not equal faster synonym.

Can you derive what effects of an instruction are used and which not from the code?

The eventual state of all data after an instruction is all that matters. It defines whether something is a synonym or not or an 'almost synonym' and which data is effected differently and how.

Perhaps for every instruction you can define the effect on any data. If you combine several instructions you still have a single description of the effect on the data. Comparing the effects on the data will give you synonyms. Comparing the effects on the data can also give you 'almost synonyms'.

-----

Sudden idea: to manually make the tables out of human readable text you can slide the (transparent) text over the tables and use the first thing in the text to shift it into the right spot in the tables. That thing then disappears from the text. Based on what's next in the text you can move to the appropriate spot in the tables and shift it in there.

Some instruction sequences might have the same result and are both the shortest and the fastests, but then there's the heat produced by the CPU. Some instructions may produce more heat than others. When it gets too hot then the CPU must be slowed down, so limiting the heat may be something else to keep in mind.

There might be optimization several priorities. You might want to have the fastest solution, you might to limit the heat the CPU produces, you might want the solution with the least disk access... I want to offer a general optimization which favors speed, but also other things to a certain extend, but I have to keep in mind that other optimizations might also be wished.

The objective is to benefit fully from assembly, but not to have to bother with the downsides of it. On top of that there's added clarity by an alternative naming and forgetting about different instruction forms.

If you take the assembly division instructions for instance then you see that certain forms are faster. Careful use of the alternative language I will make will make faster code(?). But most optimizations are done by the compiler. The amount of tricks and things you need to know to get the optimal result is far smaller than programming in assembly.

-----

Another idea: an 'overhoring' program is still useful.

-----

The only thing you have to do to really optimize your code would be to specify with each line or group of lines what storages matter to you or maybe even this can be exactly derived from the rest of the code. Consider for instance always overwriting data previously set by an operation before you ever used that data. Then that data is unimportant and the synonyms you can choose from are allowed to produce different results for this particular data.

##### Inline

Inline modules run faster because there aren’t any procedure calls. Procedure calls require things to be put on the stack, a round trip through vector tables and a return. Procedure data needs to be dynamically allocated.

Inline modules have their procedure data in a fixed spot in memory, though seperated from the code. Calls to procedures are not made. A lot of times what otherwize be the first line of a separate procedure would be the next line in code. Other times a jump is made to another part of the code.

Not making jump statements makes the fastest code. This has to do with the CPU’s architecture. The CPU fetches opcodes from memory while it processes the current opcode so that it has the next opcode ready straight away after processing the current instruction. Also, the CPU decodes the next instruction while it is executing the current instructioin. The CPU fetches the opcodes following the current operation. The prefetches operations are called the prefetch queue. A jump invalidates the prefetch queue and the decoding of the next instruction delaying the processors execution.

#### Machine Code

Also: data should not be inline of the code. Jumping over data takes time, because it invalidates the prefetch queue and the decoding of the next instruction.

If jumps. Loops. Blabla jumps.

Shorter instructions also boost the performace of the prefetch queue.

Using immediate data is fastest, Using registers is fast, using aligned memory is slower, using unaligned memory is even slower. And it’s not true. Shorter instructions use up less ‘power’ of the prefetch queue, the bus is used less and can fetch memory, maybe instructions. The shorter the faster.

The pipelining in a CPU also requires you to limit jumps in the code, because when you make a jump, the pipeline gets flushed.

Gee, what could make a processor fast you can totally break to pieces with a bad compilation/assembly program.

I am still convinced that it is possible to make a compiler perform extreme optimizations that make a program more optimized by the compiler than you could ever do by hand in a reasonable amount of time.

Read over the 886, 8286, etc. of ‘Art of Assembly’, and sum up optimization tips.

The Art of Assembly does not cover operating system instructions extensively. I must look for an additional text for coverage on these instructions.

The only reason I’m reading ‘The Art of Assembly’ is to learn how you manage the modules of the machine. To have a complete overview of in what ways the machine is controlled. I’m not learning it to learn how to write a good algorithm or to learn about a flag you’ll never use. Optimization in terms of CPU time saving is interesting, but I need to focus on what I need to control in the machine.

Art of Assembly says that segment addresses don’t resolve to a memory location arithmetically. A segment address is a location in a table specifying the memory location of the address. Art of Assembly also says that it is not up to the programmer, but up to the operating system to handle this table.

##### Machine Oriented Optimization of Code for Speed

I see that knowledge of the computer adds a lot of rules to how to optimize assembly code. I was worried that this meant that J Core could not become simple and could not set simple rules, because it should embed all these optimization rules.

A while later I figured out that this wasn’t true. The instruction set that is most of the Assembly language is a clear set of instructions to use with which you can make clear code. You don’t need to be concerned with the machine to make this clear code and with good architecture, it will operate at warp speed for most purposes. It is only when you WANT to optimize for speed that you need to understand the machine. The same should go for J Core. The programming model provides a simple and clear way to work. It is only when you want to optimize things that you should be concerned with the machine.

Assembly sets this challenge: the stuff you program has effects on the speed that are totally obscure when you just understand the code. The code is a complete cloud over what effects it has on the speed. If Intel does it, then J Core can do this: the stuff you program has effects on the speed that are totally obscure when you just understand the code. The code is a complete cloud over wht effects it has on the speed.

So a message from me to me: ‘Don’t worry: design a wonderful programming model and forget about the speed implications. Everyone makes it a difficult to do machine oriented speed optimizations. I can’t correct this and I am not going to.’

An example is accessing memory: accessing memory locations near eachother is faster. You can access memory locations far apart from eachother all the time. A J Core programmer can too. A J Core programmer can work with unbounded consecutive stretches of memory. If he uses it in a certain way he will gain tremendous speed. He can’t be aware of this just looking at the programming model, just like in assembly.

Note: when the code is clear, everything is faster. Weird time consuming mistakes are made when code is not clear. The code might be so unclear that the programmer doesn’t even know he made a mistake at all.

-----

In regular programming languages all parameter values are put on the stack. Sigma, however, decides on its own when or when not to use the stack and optimizes the use of resources.

#### Possibility Optimization

(Splines that could happen: something for another section discussing possibilities, something that also has to do with a kind of code optimization.)

#### More Stuff

#### Optimization

##### Access Optimization

The object oriented structure of the system is retained in the compiled code. This is done *only* to be able to **program** orderly with them.

However you don’t have the names of modules, classes and other elements. You need a symbol tables to tie names to the numbers that elements of the system have. This symbol table is only used for software development.

##### Speed Optimization

###### Leave Out Access

To make a system perform faster you can take away the object oriented structure. With that you gain that you don’t have to put directions on the stack and make jump statements to get to the procedure.

Another way to optimize code for speed is to inline procedures. Preferably all procedures if there isn’t a potentially infinite recurrence. (You could inline a maximum of recurrences.)

###### Inline

As long as you’re going to program with the module you need the system structure retained. The procedures need to be separate, because they can be arbitrarily called. Procedures that are called through public interfaces that is, because procedures that are private can be entirely inlined.

A full inline compilation is a nice option, though. This speeds up the system considerably, but you can’t program with the modules furtherly.

If you however keep the Symbol system specifications of all the modules then you don’t really need the compiled module structure per se. You can inline all the other module’s code into the new module’s code. This increases code size, but also speed.

That way, however, there can be no distributed systems without supplying the source code with it.

J Core’s task that always remains is to manage threads and memory. Calling conventions and system structure are an optional management. A module can be compiled fully inline in case of which it cannot be programmed with unless you have the source code. A module can be compiled with preservation of the structure allowing you to program with it if you have a symbol table.

J Core supports both of these methodologies at the same time. An inline compiled module cannot be called. It can however be run and have its thread. It can even call other modules that have been compiled outline. It can even create other threads. It calls J Core every now and then to allow other threads to run, but it has no division in procedures and it can only be called in one way: run it. The module has a place in the module vector table however. The jump statement at the module vector table record goes directly to the start of the module’s code.

An inline compiled module can create objects from its own classes, because the memory layout of a classes *are* compiled with it.

#### Logic optimization

Sometimes code is executed that can be done simpler. You can recognize constructions that can be made simpler for the computer and replace them by the simpler constructions.

#### End State Optimization

If a stretch of code results in nothing other than a configuration of objects and data, then you can store the end result of it in an end state. When that code is normally called, instead, the end state is loaded immediately. That saves a lot of time. This can only be done in cases where the end result of a chunk of code isn’t variable. If it is only a little variable, you could play with this optimization a little and store end states of the invariable things and canculate the variable part *or* store multiple partial end states, one for each variation.

In development, this optimization is not that much wanted, because the code changes there all the time. After compilation the code doesn’t change and therefore such an optimization is useful.

### Symbol to Symbol: Collapse Object Structure, Reduce to Procedures

This translation can be explained without knowing the Sigma module structure. This translation is a step in translating Symbol to Sigma, but the rest of Symbol to Sigma translation can only be explained if Sigma module structure is explained.

### Symbol to Sigma

A Sigma module contains the organization of a module as well as the machine instructions of the module. The organization is a tree like groupage. To get the tree like structure you tie together tables by referring to another table inside one table. To create consecutive memory out of the table structure you simply string up all the tables including their references to eachother. References are the memory addresses starting at the address of the beginning of the module. The tables include the eventual machine code which is right inside the organization data.

Sigma requires you to obey a certain groupage. Sigma doesn’t require you to put the tables in a fixed order in memory. It only requires you to tie the right table to the right other table. This makes the hierarchy adequately in order.

In the future I might change the code organization, but I will now take the following as an example organization:

… (encapsulation diagram of the organization)

To get the memory that is the Sigma module you can stack up the organization recursively. You can also stack up the organization level by level. When generating the Sigma module it proves to be easier to stack it up recursively.

The module is now in an organization that J Core can work with and it can run the containing machine code with multiple threads and create memory objects from the memory object layouts. Other modules can quite easily work with the module.

That’s what J Core basically does: run machine code in multiple threads and create memory objects. If you correctly organize an module then J Core can work with it.

J Core sets some rules on how to organize the modules. It doesn’t create the modules itself. It only sets the rules.

J Code executes the rules by creating correct J Core modules.

So J Core’s task is to work correctly with a code organization. J Code’s task is to create the code organization according to the rules.

#### Symbol Tables

Outline compiled modules can be programmed with. However, there are no names for the module itself, its classes, interfaces, members and arguments. Those entities only have numbers, because numbers are faster to work with. To be able to program with the modules you must have the names. As a human being you simply can’t work with the numbers. Symbol tables can be generated during compilation that tie names to the module, all its classes and interfaces and members and arguments. These symbol tables are only used when programming with the modules. When your program runs then the symbol tables aren’t used, but the numbers.

### Parsing

#### JCODE:

#### Tools om te parsen

Onderdeelomschrijvingen opsplitsen in criteria

First you split up the string into sequences of characters of the same

category. A character can belong to multiple categories.

Then you use this information to compare it to a specification containing a sequence of wildcards:

* Exact string
* Or wildcard:
  + Category: specific or not specific
  + Length: specific (?) or not length specific (\*), between lengths (? ??? or ?? \*)

- Not'ness

- Or'ness

Then a set of findings can be built up.

The findings can be extended with a replacement to correct the format of things.

Let's take this nomenclature:

!

=...

=\*...

?

\*

@

\*@

#

\*#

a

\*a

.

\*.

\_

ch.12352->

->Ch.1423452

=\*-

=>

c

h

\*\_

.

\*\_

\*n

=\*- => c h \*\_ . \*\_ \*#

->Ch . 13242

-->cH.1

~=

~c~

~h~

~\* ~

~.~

~\* ~

~\*n~

~=\*-~

~=>~

Splitting starts with categorizing characters.

Categories could be:

numeric

alphabetic

punctuation

Uppercase and accents are approached with characters categories. A a ? â á à Á À Â å a A Å ? can be seen as the same category so they will all be interpreted as 'a'.

-----

In Symbol, when you use public objects, direction is ambiguous. Also, an object in ActiveX is actually a property. Why is it a circle in my case then? Why isn't a property an object then? oh. No. Getting an object is actually a procedure, but you use it as if it's just an object. When you can set it, then that's a definite procedure. Darn. I have to revise Symbol. It's not that good this way.

-----

For string manipulations:

When you want to do two manipulations that require you to go through all characters you could use a single traverse function which you give an array of references to functions so that you only loop once, yet apply two string transformations. For instance, making the letter case of names proper and removing every third space.

More details are involved, but that's the general idea

-----

Perhaps there's an optimization to be derived from that: merging loops.

## Controls

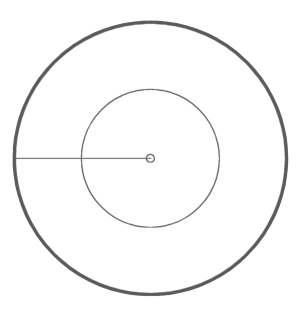
### View

The Symbol view will simply be the drawn diagram. Complex calculation is involved with placing the symbols on screen, because the user does not use the mouse to place the symbols on screen. The symbols are automatically positioned. The complex calculation is the most complicated about the view, but for calculations there is referred to another section “Calculation” so that won’t be in this chapter. What I’ll describe is the other features that the view supports.

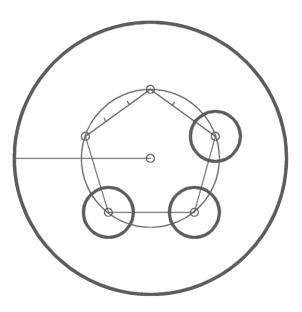
The view supports Zoom and Offset by mouse movement. It supports a map for an overview of the symbols. It can display grids and rulers. It can use color indicators to let jump symbols matching certain criteria. Coloring is also commonly used to display inheritance relations.

#### Building the Mesh

The mesh is built up from the outside inwards. There is always the first single circle: 0. The inner circles need to be laid out over a circle with a radius half the size of the parent.



The the radius of the inner circles depend on the amount of inner circles. If you draw out an n-gon connecting the centers of the inner circles together, then the radii of the inner circles are a third of the length of the n-gon’s sides. (See “Calculations N-Gon”)



The n-gon formula is:

Cos ½ (Pi – (2Pi / n)) = (½ l) / r

So:

Cos ½ (Pi – (2Pi / Children)) = (½l) / (Rchild / 2)

l = 2 \* (Rchild / 2) \* Cos ½ (Pi – (2Pi / Children)))

So the the radius for a child is:

1/3l = (2 / 3) \* (Rchild / 2) \* Cos ½ (Pi – (2Pi / Children)))

### Interaction

The interaction isn’t much more than two text boxes and something that interprets typed single word or two word commands and then calling the appropriate function of the document. Restrictions are managed by the document so the interaction doesn’t have to concern itself with protecting the data.

### Calculation

#### Pythagoras

#### N-Gon

This section covers the calculation of equal sided n-gons’ coordinates, sides and radii. This calculation is part of the JMath library and is encapsulated by the class NGon.

Without pictures the facts are as follows:

l = 2r \* cos ½ (PI – 2PI / n)

r = l / (2 \* cos ½ (PI – 2PI / n)

x = r \* cos((2PI / n + b) \* I)

y = r \* sin((2PI / n + b) \* I)

Fortunately I made pictures. Get ready for adult sesame street.

##### Radius to Side Length

I need the length



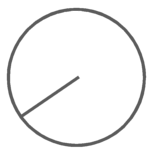
Of the sides of an equal sided n-gon



The describes a circle



And all I have is the radius



Or rather:

Got r

Need l

I need the length of the sides of any equal sided n-gon that describes a circle with a defined radius

l = 2r \* Cos ½ (PI – 2PI / n)

##### Side Length to Radius

r = l / (2 \* Cos ½ (PI – 2PI / n)

##### N and StartAngle To Coordinates

x = r \* cos((2PI / n + b) \* I)

y = r \* sin((2PI / n + b) \* I)

#### Vibration

...

## Brainstorm

###### ‘Normal’ Variables

Variables are objects. A data type is a class. Its members operate on some data.

Constants (can be assigned only once where declared (initialization))

Copy is like variable assignment

I will translate each common variable action for you to symbol object structure methods.

##### Brainstorm

Event interfaces

Multiple sink events

### Darn

#### Inline

Private procedures can be compiled inline except when infinite recursion is possible.

You can also choose to compile all procedures inline. This means that when you have compiled the module you don’t have separate procedures anymore, so you can’t have separate interface members, so no interfaces and no classes. That way the whole object oriented organization collapses and you have one pile of spaghetti code. Spaghetti code that is *way* faster than the organized code. However, you can’t use the classes of the module anymore. You’d have to have the module’s source code to be able to program with it again.

#### Symbol Tables

Outline compiled modules can be programmed with. However, there are no names for the module itself, its classes, interfaces, members and arguments. Those entities only have numbers, because numbers are faster to work with. To be able to program with the modules you must have the names. As a human being you simply can’t work with the numbers. Symbol tables can be generated during compilation that tie names to the module, all its classes and interfaces and members and arguments. These symbol tables are only used when programming with the modules. When your program runs then the symbol tables aren’t used, but the numbers.

#### Yielded over from old J Core documentation

##### Inline Module

A module can work faster if it doesn’t supply any procedures. The only call to the module then is to simply run it. All procedures inside the module are compiled inline then. However, it must be possible for the module to create its own memory objects, even though there aren’t any separate classes anymore. The inline module doesn’t have vector tables. It does have a table with the addresses of memory object specifications. On top of that it doesn’t have procedures, but instead one pile of spaghetti code.

##### Inline Module Usage

If you choose to compile a module inline then you have a faster module. This code cannot be called by other modules though. The only way to use the module in another module is to have the source code and compile it into it inline.

However, the inline module can call another outline module’s procedures. You can’t call an inline module’s procedure, but an inline module can call an outline module’s procedures.

-----

I’ve gone on about the assembly side of it, now the easier interface.

<

Even though it seems that all sorts of stuff can happen at the same time and all sorts of objects and code can work together in an unfollowable turn taking. This is not true. One thing happens at a time. One thread runs at a time, one procedure runs at a time, one data structure is used at a time, one module is running at a time, one class is running at a time

>

The only way you want to work with J Core is to design modules that run under J Core. It is about is designing modules. Module design is programming so it continues in J Code. In J Code you can design modules in a simple matter. They can be compiled to machine code modules that can run under J Core.

So the easy way is not J Core. J Core is not one of those modules that has easy interfaces because at a level this low there are not modules yet.

Module design continues in J Code.

J Hardware requires you to design modules writing assembly language J Core procedures.

For J Data you can write the procedures in a higher level language, which nonetheless translates to assembly.

I need to make a higher level language that does not perform tasks of the modules, but provides just the neccesary to specify automation between the modules.

##### New

The lowest advisable level to program J Core modules in is a Symbol document. This specifies the system structure. The procedures are also specified in symbol. The symbols in a procedure can refer to the Symbol Assembly library or can refer to other procedures inside the module or to procedures outside the module. The procedures can be displayed as a diagram, but also as a language much like Assembly language and OO Basic intermixed.

J Code will convert the symbol document into the tables specifying the classes and interfaces and translate the procedures to Assembly code that makes J Core calls and obeys the extra set of rules that J Core sets that you don’t have to worry about as a programmer.

Management of segments, optimizing code for speed, etcetera, are all controlled by J Code and I don’t want programmers, including me, to have anything to do with it, because it is possible to automatically do this. I think real assembly is not worth manually programming with and should be automated.

-----

Keep believing in Symbol.

Again I am analyzing something I wrote for a complex problem (a problem that others made complex for me), and again I wish to know what calls what, how many times something is called and where. When I'd have a symbol view of the code then I'd instantly see everything.

### Usage of J Data

Storage lists in Symbol should be able to be standard arrays.

Shouldn’t I create my own simple array class? Yep, it won’t work under Microsoft, but at least the code is structure in accordance with Sigma… or something.

Oh, huh? How can I create an array without a base data structure for it? Huh?

Array1 = New Array(Integer)

Array1.Items

Array1(3)

-----

Operation is actually a nice synonym for a function, procedure, subroutine, routine, etcetera.

### J Code

I have to think about and write something about the fact that most properties that can be written are hardly ever read.

For clarity, there are cases in which common parameters of members are best included in each member, but there are also cases in which common parameters of members can best become members of their own. For instance in set of members with a lot of parameters that have the same values a lot. If the values of the common parameters are the same most of the time, they are best made members of their own.

If the values of the parameters are different all the time, it's best to keep them as parameters.

This is the contemplation for clarity, which is something different from the contemplation for efficiency. I wonder if the stack (with random access) is really faster than regular memory access. That's the criterium for efficiency contemplations.

##### J Code and Code Style

Je moet om properties te groeperen in Vb een hele set klassen maken.

Eigenlijk zou alle logische genetiviteit gegroepeerd moeten worden, zonder dat je er een klasse van hoeft te maken. Maar in Symbol maak je er al gemakkelijk gewoon een klasse van. Het groeperings symbool doet dat eigenlijk al. Anyway, als je een Button List hebt, dan moet je per button iets instellen Button (aIndex) . Visible bijvoorbeeld. Je kunt een klasse maken van Button Lisst zijn Button. En dan een array van die klassen maken. Maar kan je niet gewoon Button (aIndex) . Visible als property van de Button List declareren? Hmmm… Bij de wee, ik moet naamgeving uitleggen ergens. Ik heb bij de wee iets veranderd aan mijn naamgeving. In Button List, bijvoorbeeld, had ik ooit een Button Visible (aButton As Long) As Boolean, maar ook een Button Width As Double. Maar die laatste was hetzelfde voor alle buttons, dus niet een Button z’n width, maar de Buttons hun width. Buttons Width dus, niet Button Width. Gepuncteerd:

Button (aButton) . Visible en Buttons . Width. Dit voorkomt dubbelzinnigheid. Some is het ook zo dat je een opeenvolging van namen als identifier hebt: Group Supplier Type References. In dat geval is het 1 Group Supplier Type’s References, want de laatste naam is het lijdend voorwerp. Als het om een Group Supplier zijn Type References gaat, dan weet ik even nog niet exact de naamgeving, alleen dat het anders moet zijn. Group Supplier Types References? Omdat het om al de Types van een Group Supplier gaat? Maat een Type Reference was in die database 1 entity geworden, dus om nu verstoring in die naam aan te brengen… Het is een Group Supplier’s Type References, maar… hmmm logisch gezien is ‘Types References’ wel goed.

Meervouden kunnen by de wee dus verzamelingen aangeven, maar ook een aantal.

Het gaat dus om het genetief stapelen van namen en het juiste gebruik van enkelvouden en meervouden.

En ook 0 based tellen, maar wel arithmetische aantallen.

Misschien nog wat over het vermijden van loops.

-----

Sometimes there’s a distinction between a class that serves as a simple storage for, let’s say Object Interfaces, and a class that combines other classes ondergeschikt aan Object Interfaces and the storage class I mentioned. In both cases you could use a plural to denote you manage a collection of Object Interfaces. To distinct the storage list from the complexer abstration class you call the storage class Object Interface List class, while the complexer abstraction class you call the Object Interfaces class

If you use constructions like described in the last paragraph, then if a class is an abstract of a basic data structure storage, but the entities involved have no child entities for the storage to be combined with, then the class doesn’t have to be called a list, because it manages everything up until this abstraction level… < I need a better description for that, man >

-----

List of objects: Object List. No plural used. It could mean Object’s list, so it should actually be Objects List. Hmmm… It don’t sound too great, but…

A data abstracting class should be named after its main dimension / main entities / target entity. < In an index, the source entity should be mentioned too. >

I took back using a plural to identify a collection. I can’t tell amounts and collections apart that way and code became unreadable to me.

I also called both storage classes and combining classes ‘lists’ now.

Properties vs. parameters. Consider named properties. Make a deep comparison of properties and parameters. I still don't know the speed of the stack compared to a normal memory address.

To get an object (property) structure nicer, so that entities are actually objects and not just properties with a common prefix, you can do the following:

You create a class maintaining the storage structures of the entities combined. Then you create another class that refers to this storage class, but actually represents only one row in it. Then you can use this class as the type of a property of another class and this property has nice member access to the object. I actually think this refinement of my general structuring is very required. Only then do the interfaces look as what they represent.

This way the following:

ModuleInterfaceModuleInterfaceMemberType(4, 2, 3, 21)

becomes:

Module(4).InterfaceModule(2).Interface(3).Member(21).Type

or in Symbol:

Module Interface Module Interface MemberType [4, 2, 3, 21]

Becomes:

Module [4] . Interface Module [2] . Interface [3] . Member [21] . Type

A much better representation of actuality, and readable, unlike the original structure.

Member access isn't only useful in, also to group things. To get member access to the grouped things you actually require another class, which is unfortunately not a child class in visual basic, but another entry in the richly occupied class list in Visual Basic. Would you have a thing as child classes, this uglyness could be omitted, because it would then be a tree structure. Another thing with grouping properties is that when you put them in a child class, the child class doesn't have access to the main class's members just like that. You might want the grouped properties to interact with the other properties. In that case you should have to use another phenomina than just a child class. Something solely made for

grouping. In VB you don't have this. For that, for interaction with the properties outside a class that groups a few can only be accessed if you give the group a reference to the parent class. This adds a method or property to the group that we don't want to see outside the main class. We can't verhelp this inside the module. But outside the module we don't need to see this member that sets the parent class. We can hide them from outside the module, by declaring them Friend, instead of Public. But inside the module we will still see it as part of the member of an object property.

#### Other

In the Windows API documentation somewhere it is told that delegating specific windows message handling to other functions in the window procedure will prevent the stack from overflowing when using many local variables.

However, if you make it so that the local variables are allocated on the stack imperatively where they are declared, when the declaration code isn’t reached, this allocation doesn’t take place. Microsoft people might advise (in documentation about VB) to put local variable declarations at the beginning of the procedure. So it may not be the best methodology that makes the argument for this. It’s may not be a good idea.

It may not be a good idea regarding the using a function a message. It is however so that variables on the stack aren’t freed until you exit the function, so you can’t free declared stack variables. Maybe I could free it in J Code if it would be required.

-----

Windows often stuffs a lot of booleans in a bit field. In the Windows Controls, often styles are used, which are booleans in bit fields that you can switch on or off. You supply the bit field at creation of the control window. You can change the bit field of some bits in the fields by sending window messages to the control.

I would just make them properties most of the times. At most I make them creation arguments. Supplying all as creation arguments is no better than not initializing the control and letting the programmer set the properties. Properties are not applied to the representation straight away, because display operations are postponed, because they do not affect control logic data and operations, yet cost CPU time.

What is better in the Windows way? If anything, can I embed it into my way, without losing any benefits of my way?

-----

Windows might stuff multiple values into a single data unit. For instance, a bunch of boolean properties into a single integer as a bit field. That is actually size efficient. Nevertheless if you want to change one value, it requires you to retrieve the original set of values, change one flag and then assign all values at once again while you only wanted to assign one of them. The situation gets worse when not just working with a bit field, but working with a structure that contains many properties that can only combinedly be retrieved and assigned. Then there isn’t really a space efficiency and the amount of data transported is larger and more ifs are involved and more code, because handling each individual value requires its own code anyway. All ‘overhead’ given by using separate values seperately is that you have more callable functions, which is actually more a benefit than an overhead. The code handling the value assignments is there in both situations anyway.

-----

Consider making designers for certain things. If a designer would help.

-----

The simplest way to turn a variable into an expression is to enclose it in child-heses.

-----

Tip If you're going to use the value of a property more than once, your code will run faster if you store the value in a variable.

Conversely in symbol when a property returns explicitly a variable, then you compile it to be an access of the value.

-----

Each time the text changes in Text1, the child form's text box Change event sets boolDirty to True.

Not to have the setting of the variable execute all the time, you can make two procedures one of which sets the variable, the other of which doesn’t. After the variable is set, the function pointer currently pointing to the function setting the variable is changed to pointing to the function not setting the variable.

Wow another kind of optimization

-----

* Nominative: object or member call
* Genitive: member access
* Dative: parameter
* Accusative: assignment

-----

Eventually I want to be able to program microchips. Create code out of which you create microchips.

I eventually want to be able to program robots, machines, objects, mechanical things. Physical things. Not make microchips to go into them, but program code that describes the mechanical thing and have other machinery create a machine out of it.

Code

When symbol code is compiled to code executable on Sigma on x8086 machine, extreme optimization is used that is exact and mathematical, making the module fast, small and usable.

Code

Semantical simplification