NETE - New Era Text Editor
A Visual Diagrammatic Text Editor

Tinashe Gwena[1]*

Abstract
Computer programming is currently an exercise filled with cognitive overload. Most typical programming tools create little to no opportunity to communicate design and intentions to software practitioners who may encounter other people's code. The observation made here is that the extensive use of text based computer languages exacerbates the problem.
I am of the opinion that computer languages carry a lot of capability in current designs, but this tends to be overshadowed by excessive presentation of text. Graphical paradigms are envisioned to provide a less taxing way of designing software. Therefore, a means of creating diagrammatic views of computer programs is presented in the form of New Era Text Editor (NETE).
NETE allows the creation of higher level diagrammatic representation of the textual code, thus enabling the manipulation and exploration of the design in a multidimensional and hierarchical plane.

Keywords
programming cognitive overload — graphical overview — visual programming — code communication — intentions — code comprehension

[1] *Graphical Text Editing Initiative*
***Corresponding author**: https://github.com/ashe-cuena/nete

Contents

1. Introduction

1.1 Text Based Programming

Difficulty in communicating ideas and cognitive overhead are realities in computer programming. Be it in learning to program[1] or in maintaining existing code[2]. The source of these issues probably lies in the way in which software projects are conducted, since the root of most programming is in text based programming languages. This text base of programming then builds into text structuring. This higher level structuring is unfortunately not efficient at managing complexity[3].

Programming languages are built on the basis of natural spoken languages. Some correlation between how spoken languages and programming languages operate on the human brain have been shown[4]. Some attempts at using psychological methods to improve language design have also been advocated[5][6][7]. Beyond that, it's a matter of speculation in the history of programming as to the comprehensibility of that approach on a larger scale[8]. The hope could have been that natural human intuition and communication methods could be harnessed and re-purposed effectively in the computer industry[9]. Imaging research using functional Magnetic Resonance Imaging (fMRI)[10] though, has shown current computer programming practice operates on completely different areas of the brain as compared to natural spoken language.

This approach of text based programming works fairly well at smaller scales and smaller programs are easy enough to grasp and understand. As projects grow, however, this approach fails to scale. At some certain size and complexity software may become slow, unreliable and may even fail[11].

Since program code's ultimate eventuality is machine execution, for a programmer to spend a lot of time comprehending code, they'd be performing a feat that is difficult and might even be unhealthy[12].

1.2 General Programming Observations

Part of the hope in the development of computer languages is the enabling of humans to create and disperse computer solutions to be understood by other people[13]. Unfortunately, many other pressures seep into the actual practice of programming and then most programs end up being written simply to feed the machine[14][15].

The user of programming tools often has to go to fairly extraordinary lengths in order to communicate their intent since most programming tools often present few means to express structure as intended by the practitioner[14]. Therefore when reading through a program, one can pick up the operations which the computer was meant to follow, but not the intentions which the programmer would have had in their mind. The intentions can only really come in as a set of comments or a separate document which describes the program. This will spell a double job for the programmer and will either:

1. not be done, or
2. not quite correspond.

Figure 1 shows popular software representations which suffer from this problem of separately documenting a project. Unless the programmers are highly diligent, divergence is inevitable. This difference is often termed the Model-Code Gap[16]. Universal Modelling Language (UML) is a highly popular modelling language, but it presents something of a learning curve in its notation[17].

2. Programming Improvement Ideas

2.1 Structure and Intentions

Whenever a programmer creates a software solution, they use a mental image embedded in their mind[18]. In most environments, the discovery of this image by a third party can only come from learning the program code. More experienced programmers would use chunking based strategies to search through the code over time in a typically trial and error approach intended to eventually uncover meaning[19]. For larger projects this can be a huge, if not impossible task.

It has been found that most programming time is spent reading and maintaining legacy code rather than just writing code[20]. Therefore communication with other people, i.e. writing readable code which is easy to understand and follow is of utmost importance. Creating a platform that encourages communication is central to meeting this requirement. Finding a way to make intent explicit and clear could greatly reduce the amount of investment required to understand computer programs. Allowing the programmer creative license in expressing this intent may also be beneficial because capturing the state of mind of the designer is part of understanding their vision[21].

2.2 Graphics

Imagery and graphics could be a possible way out of the quandary[22]. Graphical design paradigms may save programmers from the issues around cognitive overload[23]. The reasoning behind this is that imagery encourages the recipient to explore the information before them. They may not have to internalise too much information before deriving meaning because the structure, trends and image are already presented to them. The less cognitive overload a person suffers, the more they can comprehend of the information presented to them.

Graphical approaches enable the depiction of different levels of detail at different scales[24]. This would be a similar mechanism to how a map can zoom out to view the entire planet at once, yet it can zoom in down to street level or even greater detail.

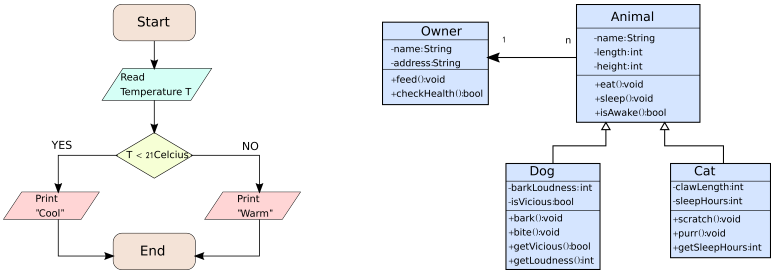
Another advantage of a diagrammatic approach is that it can be traversed in multiple axes. Text typically only moves in the direction of convention of the writing type, example; Latin based script would be left to right and then down, in something similar to a raster scan. Thus, diagrams afford the viewer random and immediate access to elements, so depiction is inherently multidimensional.

3. Existing Systems

3.1 Typical Visual Environments

Visual environments are generally found in the two classes which are Visual Programming Languages (VPLs) and visualisations. The difference usually being that:

1. VPLs encompass the syntax and other programming structures completely in a graphic environment. This would mean the creation and editing of the program



(a) A flowchart example **(b)** A UML class diagram

Figure 1. Software graphical representations

is done graphically, although sometimes, direct text editing may also be possible.

2. Visualisations create a graphical interpretation of an already existing computer program[25]. This may be done in order to enhance understanding or for the sake of analysis. Editing and execution are typically conventional processes with visualisations.

3.2 Visual Programming Languages

Some typical examples of VPLs are:

1. Quartz Composer from Apple[26], Blueprint Visual Scripting for Unreal Engine[27], Labview from National Instruments[28] and Node-Red originally from IBM[29]. These are the typical VPLs which are built on the notion of nodes and arcs. Nodes are operations and arcs represent data flow. These types of environments are quite popular in multimedia and engineering. A limitation which may exist with these is the number of nodes and arcs that users can comprehend from the display[30].

2. Scratch [31] is a popular learning environment commonly touted as a VPL, but in the strict sense it's a regular text programming language with visual elements overlaid to enhance comprehension. Apart from that, the ultimate intention of scratch is to provide a gentle introduction to regular programming so that younger learners can grasp an understanding, but with the intention of "dropping the crutch" at a later period when the student has become more confident.

3. BlueJ [32] is built on an interpretation of UML class diagrams. It gives a means of drawing object relational diagrams, then it generates the skeleton Java code underneath which can then be edited in the embedded editor and run.

3.3 Visualizations

Examples of visualization systems are:

1. Designite [33] is a design quality assessment tool. It is built to provide a detailed metrics analysis. Above that it helps identify issues adding to design debt and assists in improving software design quality.

2. SolidFX [34] is a reverse engineering environment built to assess the maintainability of C/C++ code bases. It was designed to support code parsing, fact extraction, metric computation, and interactive visual analysis of a given code base. Some of its listed abilities include: finding complexity hot-spots, assessment of modularity, assessing maintainability and tracking copy-and-paste patterns.

3. Sotograph (Software-Tomograph) [35] uses architecture models defined in Sotoarc for additional analyses. It brings with it tools that allow for detailed structure analysis, quality and dependency analyses on different levels of abstraction. It helps identify issues such as cyclical dependencies and duplicate code blocks. This tool assists in analysing a large amount of quality metrics and time series metrics, the latter which which are useful for monitoring trends.

3.4 Comparison with NETE

NETE is neither a visual programming language nor is it a visualisation. NETE is a means of editing and organising text. Though visual, it is not a visualisation, because the graphical elements form part of the editing exercise in putting together a solution.

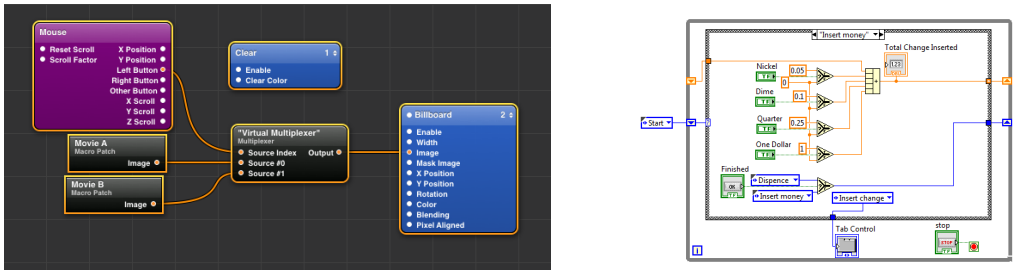
Therefore it is not a means of deriving metrics for the purpose of analysis. It is more of a means of creating illustrations that communicate the intended structure of the underlying text. This editing method then forces a tighter coupling between the model and the code, thereby reducing the incidence of the model-code gap.

Creating such a visual structure should then reduce the need for a mental image and reduce cognitive load by effectively managing complexity.

4. Theoretical Concepts

4.1 A View of Text - OHCO

The notion of structured text brings with it the idea of an Ordered Hierarchy of Content Objects (OHCO)[36]. Although



(a) "Quartz Composer", Mindoftea, CC BY-SA 3.0
https://commons.wikimedia.org/wiki/user:Mindoftea

(b) "LabVIEW", CC BY-SA 3.0
https://en.wikipedia.org/wiki/User:Smith_Pohl

Figure 2. Visual Programming Languages

this is a somewhat contested viewpoint, for the purposes of computer programs and for the purposes of this project, it's a useful notion to use.

4.1.1 Order

Order tends to be inherent in computer programs. Statements are executed one after the other and the order of this execution influences the desired outcome. Sometimes though, order exists unnecessarily. The nature of text is that it is serial and thus elements have to follow each other, even if this has no influence on the outcome. An example could be declarations of functions. Order is absolutely necessary in the program in some areas(either some small algorithmic implementation, or some necessary once off declaration e.g. a global variable).

Therefore it is a useful notion to apply order when necessary, but it would be good to reorder program elements when and where it could bring in other advantages such as logical grouping.

4.1.2 Hierarchy

Then there is a notion of hierarchy. Normally this would mean that we break down the text into its constituents and then certain markers in the text will bind some other text to contain it into some kind of hierarchical structure. In the case of computer programs this could be the definition of a function or an object will contain certain items, within that contained code there may be loops or conditional structures that may exist. These can form natural structures which most users can see. A problem though, is that there may be underlying subdivisions which are not visible, but are part of the programmers perception. Perhaps a section calculates a known equation or subroutine, yet there is no syntactical structure to highlight that.

Such arbitrary substructures may be useful for a programmer to treat in isolation. Other invisible hierarchical structures which are useful are conceptual groupings. Perhaps there are a group of functions which bear some similarity in application. It is typically not easy to group these in a visible structure that relates them. Furthermore, there is typically no means to group these invisible conceptual groupings into even more higher order groupings.

4.2 Memory Operation

4.2.1 Chunking

Chunking is a mechanism in which the brain can manipulate large information sets in working memory[37]. This happens by amassing sets of possible scenarios over time and storing them as sizeable chunks. Example of this could be an experienced chess player. Over time they would have encountered a lot of common possibilities which they memorize in large groups of short sequences, thus forming chunks.

A similar strategy is hypothesised to work in seasoned programmers. Over time they accumulate large chunks of common design patterns which they seek out when deciphering new code[19].

4.2.2 Grouping

Grouping is a strategy using during information dissemination which provides a strategy that can be likened to pre-chunking information. Grouping related bits of information provides a means for a recipient of information to better relate elements and thus speeding up the learning process[38].

5. Implementation

This project attempts to merge the propositions and theoretical background that have been stated. It is viewed as a way in which the scourge of text based programming can be tamed without throwing out the baby with the bathwater. A lot of work has been done over the decades to advance programming languages to produce most of what is available in the programming world today. Thus, it would be shortsighted to attempt to forego all that current and past programming has to offer in an attempt to buck the norm.

5.1 NETE

NETE, meaning New Era Text Editor presents a different way of viewing those same programs by creating an alternative view of the text. What this does is to enable a graphical approach to creating and editing text. Graphical here is capturing two senses of the word i.e.

1. Drawing pictures on a screen,
2. Mathematical nodes and arcs.

The entire proposition is carried by the two elements i.e. node and arc, and how these are creatively combined in a project.

5.2 Summary of Strategy

The approach here is to reduce chunking based strategies and mental imagery that are mentioned in Section 2.1. These strategies require time and experience and thus carry a high cognitive overhead. Instead, this work is promoting visible grouping and explicit structure i.e. visible imagery, similar to Section 4.2.2.

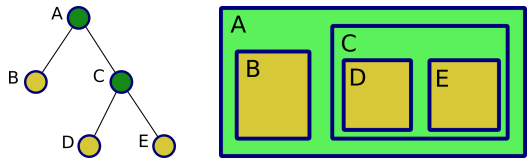
Using OHCO principles mentioned in Section 4.1, the practitioner can break text into logical snippets. These form content objects. These content objects are carried as codeboxes. Codeboxes can be ordered using arcs and priorities. These codeboxes can then be placed in containers to form hierarchies. The terms *codebox*, *container*, *node*, *arc* and *priority* will be given deeper treatment later in the section.

These nodes and arcs combine to form a visible image which would reduce the necessity of a mental image. This structure also forms a sense of a model, yet it is tightly intertwined with the content such that it greatly reduces model-code gap as stated in Section 1.2. A practical example to illustrate this approach is Figure 11 which is found in Section 6.2.

5.3 Design Considerations

The design basis for this project is 2 dimensional. 3 dimensional design presents difficulties with visual occlusion, which will detract from the main goal of clarity and communication.

Representing text as OHCO is usually done using a node-link style tree structure as in Figure 3a. This conventional tree representation works by means of nodes and arcs with lower down nodes in the hierarchy branching off from the higher level node. Node-link representation is usually great at depicting hierarchy.



(a) A node-link tree diagram **(b)** Equivalent nested-containment diagram

Figure 3. Tree Representations

The main problem with a node-link representation is that it is not space efficient. Positioning of elements is usually fixed and thus does not communicate extra relational information well[39].

Nested containment as shown in Figure 3b is an alternative tree representation that has certain advantages for this application. Containment is space efficient and positioning tolerant. Visually this gives great advantages in terms of grouping and positioning elements to form logical relationships.

The basic design element in this solution is a box container. Boxes are naturally space efficient because they fit well with each other and within each other without needlessly occupying extra space. Boxes are also well suited for computer screens which are mostly rectangular, thus allowing elements to fit well within the display as well.

These containers have been designed to create slight distancing at the borders. This allows the viewer to distinguish easily between the different elements available.

Positioning is liberal and sizing is free, this way the user can formulate a visual representation of the underlying solution which best suits their intentions.

5.4 Nodes

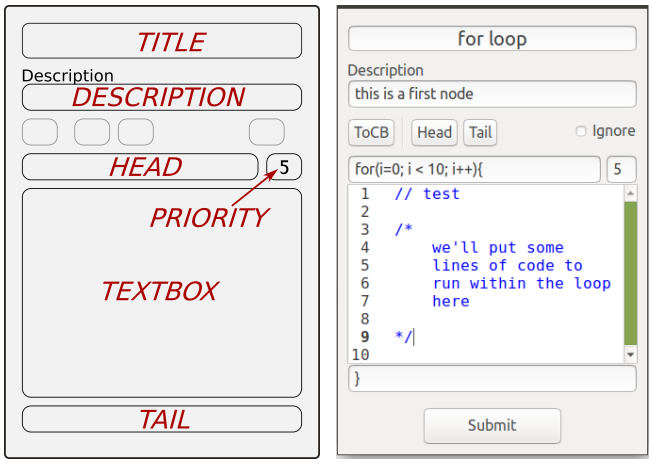
The nodes carry the bulk of the work in making all the magic happen. The same node is used in two scenarios:

1. To carry text (where it's termed a codebox),
2. To carry nodes (where it's termed a container).

The same one node does both jobs but in different scenarios. The determinant is whether or not it contains another node and this difference shows up during rendering.

Nodes also carry with them two other visible features, which are title and priority. A description field is included and this is more or less a comment feature with no real consequence on operations.

5.4.1 Codebox - Head, Tail and Textbox



(a) Codebox Structure **(b)** A populated codebox

Figure 4. The codebox

On the inside of a node there are three other important text fields, which are the head, tail and textbox. These are pertinent for the actual low level of operations and they form the basis of the text. Upon rendering a codebox, these elements are batched in the order:

1. Head,

2. Textbox,

3. Tail.

The head and tail are very important for containers because they allow an operator to be bound to the contained nodes.

The head and tail can also be useful in helping maintain codebox scope, because they allow mandatory fields to be taken out of the core code at hand and be maintained separately. An example could be a "for" statement where the head would contain

```
for(i = 0; i < 10; i++) {
```

and the tail may have "}" then the body is in the textbox. That way the programmer can focus their attention better on the body without being distracted by the syntax of the "for".

The textbox is meant to carry small manageable snippets of code. This is where the bulk of the text is carried in the project. The size of this snippet is up to the discretion of the programmer. The idea here being that it must be a simple enough section to read through and understand in a short space of time.

5.4.2 Container

In the case of a container, every aspect of the codebox applies, except that the textbox is ignored. This is because the role of the textbox is taken by the contained nodes and these contained nodes are rendered recursively to eventually produce the contained text.

Containers may act as passive holders of nodes purely for logical grouping or they can apply an operator to those nodes that they contain. An example of such could be a "for" loop, a "while" operator, an "if" statement, e.t.c. This comes about because the head and tail give containers the ability to hold scope over the nodes that they contain.

5.5 Arcs

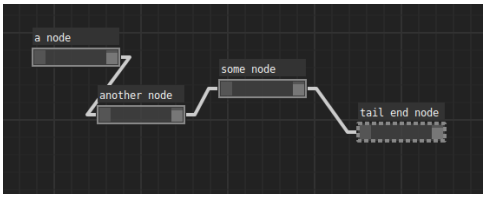


Figure 5. Nodes joined by arcs

The arc is present to allow a follow sequence to be determined and it outputs from the right of a node and inputs into the left of the follower.

5.6 Putting it Together

The relationship between codeboxes, containers and arcs then comes together during rendering. Here a few simple rules determine the outcome. Text in the output comes from three sources, which are:

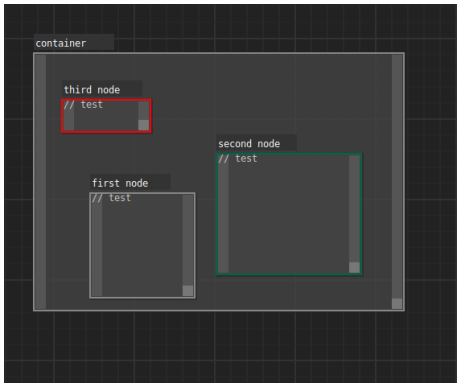


Figure 6. Nodes of different priorities in a container

1. Heads,

2. Textboxes,

3. Tails.

When the system renders a node, it begins with the head. Then depending on whether it's a codebox or a container it will render the textbox or the contained nodes recursively and lastly renders the tail. In order to render a node it has to pick the node and have a reason to move to the next node. Everything starts from the first node. The first node is found at the highest level of nodes i.e. nodes which are not contained by any other nodes. Nodes at any equal level in a container are then ordered by priority and ordered by followings (i.e. arcs).

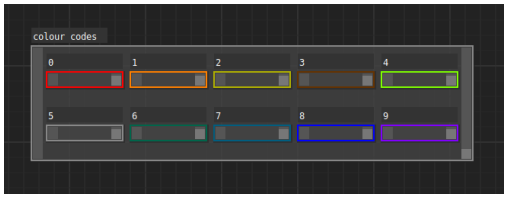


Figure 7. Nodes set to show the colour coded priorities

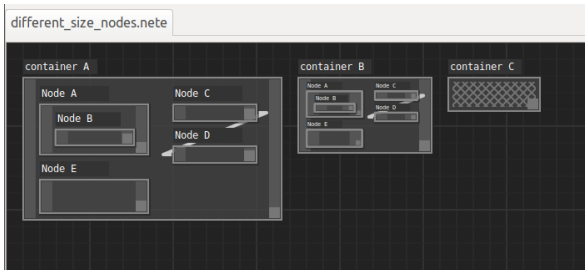
The nodes may have 10 priorities as shown in Figure 7, so this feature must be used sparingly. Priority 0 is the highest and 9 is the lowest. With nodes at the same level the highest priority is rendered first and the lowest is rendered last. The priority is visible as a colour code on the frame of the node. Nodes which have the same priority, at the same level of a container can be considered to be picked in random order(useful for parallel thinking).

Arc following is the other rule which determines ordering or rendering. Here the system will follow the arcs in its path. Using this simple set of rules, the system will stitch together all the text contained in the nodes and give out a rendered text file to be used as desired.

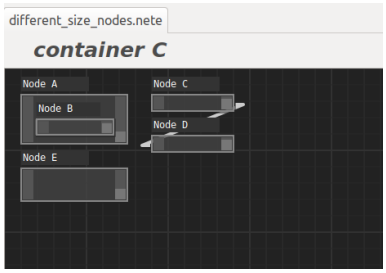
5.7 Focus and Ignore

5.7.1 Focus

This is a function that is used on containers to have elements contained in a selected node being the only visible ones on the screen, thus putting them in focus. It is useful whenever attention needs to be placed on a particular container.



(a) Different size containers



(b) Container under focus

Figure 8. Resized containers and focus

A situation though where "Focus" is imperative is a case where a container has been shrunk to a point where the nodes contained become hard to see or where NETE has determined that they are too small to render. At that point, the inside of the container will have a hatched pattern as in Figure 8a. This pattern reminds the user that the node is a container and has elements which have become too small to see. In order to view details on such a container either it have to be expanded or it will need to be focused. When in focus mode, the top bar changes to highlight the name of the container as in Figure 8b.

This feature allows the user to drill into deeper and deeper detail as and where necessary allowing control over the amount of attention needed in different scenarios.

5.7.2 Ignore

"Ignore" is a tick box that can be selected on any node, be it a container or a codebox. Once selected, the node will be struck out visually with an "X" as in Figure 9.

A node which is set to "Ignore" will have its contents ignored on the text output rendering. Therefore it serves the same purpose as commenting out lines of source code.

6. Usage Examples

As stated before, NETE allows the manipulation of text in a graphical form and this gives the text "shape and form".

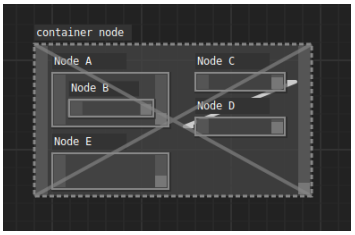


Figure 9. Node to ignore

This strategy allows the viewing and editing of "closer to physical" objects with an explicitly shown relationship. A number of real world applications have been built utilizing NETE's graphical system to illustrate its usefulness.

6.1 SMS API

A first example shown in Figure 10 shows an Application Programming Interface (API) created for an Short Messaging Service (SMS) server. The underlying language used is PHP Hypertext Processor (PHP) with a micro-framework called Fat Free Framework.

The view shown has the underlying text details hidden away and the elements making up the project are prominent. The relationships between these elements are illustrated by how they are contained and how they sit in the same container. The priorities of these nodes are colour coded. As mentioned before, the exact code underneath does not distract the overall flow and intention of the programmer's vision.

The code shown renders to ∼800 lines of code. In a conventional text editor, that many lines would not be visible at once and the relationship between different sections of the code would not be easily visible, but as shown in the figure, NETE creates such a possibility. The layout is visible in one glance on an average laptop monitor. A recipient of this project can easily glance over this overview within minutes if not seconds and get a good feel of what the programmer was after.

6.2 This Document

A second example illustrated is this very document in Figure 11. The text is set as a LaTeX document and being a text based language, it lends itself well to the NETE treatment. Small logical sections carrying a paragraph or two are carried by the codeboxes. These in turn are placed into containers according to the subject matter and the figure shows how the discussion comes together as a diagrammatic overview.

6.3 HTML Form

Figure 12 shows a HyperText Markup Language (HTML) form built for a website

The notable feature in this example is the way in which the NETE form of the code closely resembles the corresponding output from the website.

This illustrates the versatility of a graphical system such as this. A third party would probably expend lower amounts

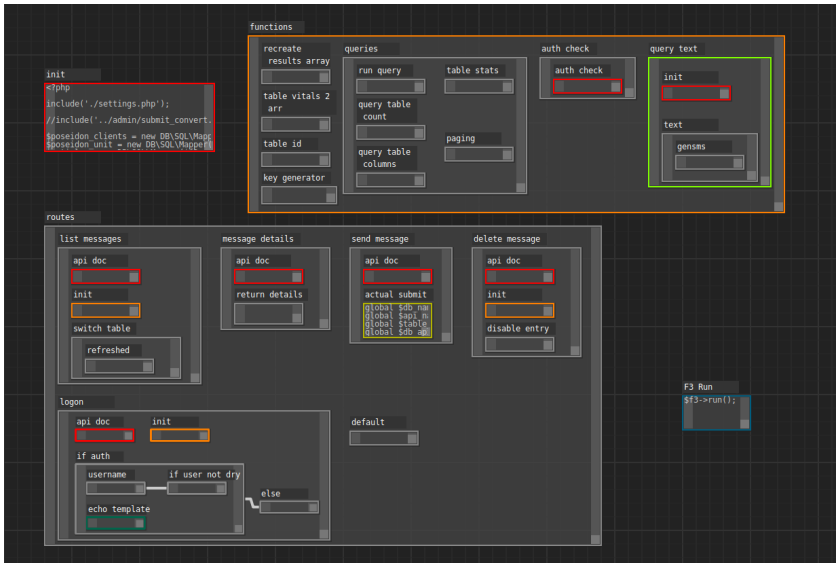


Figure 10. API for SMS

of effort matching the diagrammatic view and its intended purpose.

As a contrast, the actual HTML rendered from the NETE code is shown in Appendix A.

6.4 Issues

A few good examples have been illustrated above and NETE works well with general purpose text representation. It should be noted though that NETE does not currently work particularly well with languages highly dependent on white-space indentation as part of the syntax, such as Python. The reason being that Python is heavily dependent on the the programmer or reader of the program being able to visually compare scope created by indentation over larger sections of code. Since NETE breaks down code into smaller logical blocks, it becomes difficult to manipulate this indentation if all the text isn't visible at once.

A possible compromise solution to this problem in Python's particular case, is to utilise a pre-processor such as Bython. Bython allows Python code to work with braces as "begin" and "end" delimiters in a style similar to C language format. That is one possible alternative to the issue of Python's need to visually verify scope.

7. Conclusion

Overuse of text has been highlighted as a possible problem area in the communication of intention in computer programs and also a probable source of cognitive overload for computer programmers. A simplified probable mechanism of how text and language operate on the brain, and how this may probably be to blame for the difficulty in comprehension of a computer program has been brought forward. The attempt of this creation is to try a slightly different take on solving the problem by adding graphical elements to text editing.

Existing computer languages currently offer boundless possibilities to software developers and thus the role of text in putting together a software solution is not being done away with. Instead, the constant visibility of the text front and centre is being criticised as a big distraction for a practitioner since they will quickly find themselves not seeing the forest for the trees. Therefore it must be noted that NETE is a structural and pictorial enhancement to conventional text representation. Ultimately, this also means it is also applicable to multitudes of other scenarios where text is used, be it technical documentation design, legal document layout or even a movie script. This can happen since it gives a visual and diagrammatic approach to generic text editing. It does this in a way that hides away the underlying text and allows the user to move to a two dimensional graphical view that facilitates navigation and exploration. It is envisioned to allow conveyance of intent and flow in programming and scripting because it hides away low level text which mostly distracts users from the overall design. Instead, emphasis is placed on the overall structure of the project.

When using this system, opening a codebox for editing would probably be akin to pulling out a lens or a microscope to view issues at greater detail. Closing off that detailed view allows the user to revert back to a zoomed out view where they can deal with the overall layout.

The ability to recursively contain nodes allows programmers to be infinitely creative about how different ideas and intent group together logically in a way which is easily visible and simpler for third parties to follow.

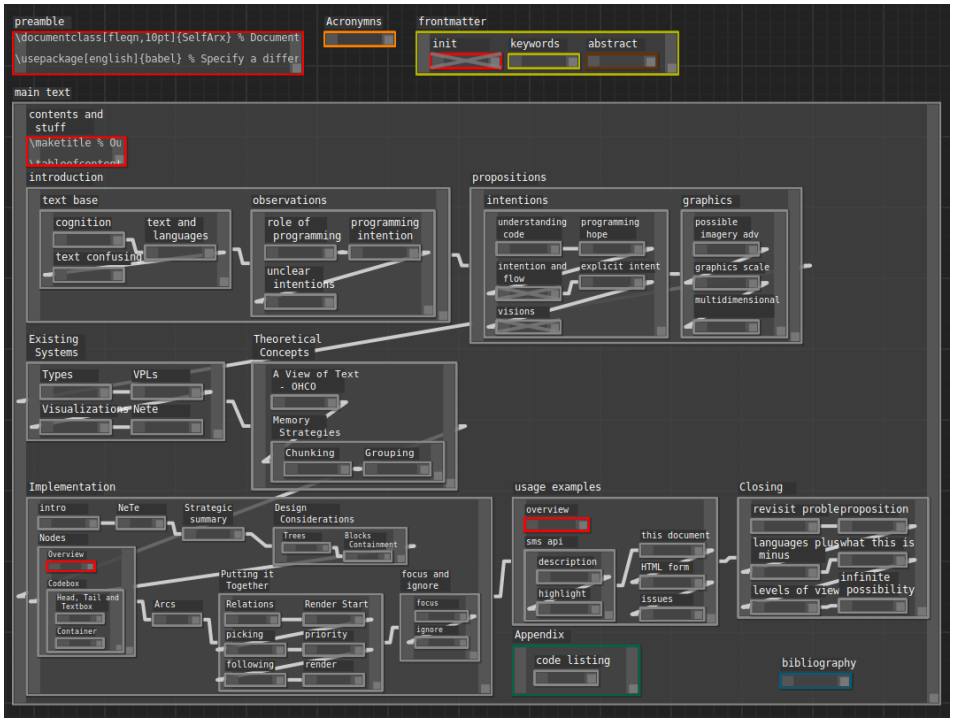


Figure 11. This very document

A. HTML Form Code Listing

```
<form method="POST" action="notesubmit" id="note_form">
<div class="w3-card-4">
<div class="w3-container w3-green">
<h3>Note Details</h3>
</div>
<table style="width:100%">
<tr valign="top">
<td valign="top" width="50%">
<table style="width:80%">
<tr>
<td> <label for="notetype">Note type</label></td>
<td> <input type="text" id="notetype" name="notetype" value="{{@type}}" > </td>
</tr>
<tr>
<td><label for="note">Note</label></td>
<td><textarea id="note" name="note" rows="4" style="width:100%" style="font-size:
      10pt">{{@note}}</textarea></td>
</tr>
<tr>
<td width="20%"> <label for="time">Event Time</label></td>
<td> <!--input type="text" id="time" name="alarmtime"-->
<input type="datetime-local" name="logtime" value="{{@logtime_corrected}}"> </td
    >
</tr>
</table>
</td>
<td valign="top" width="50%">
<table style="width:80%">
<tr>
<td> <label for="user">User</label></td>
<td> <input type="text" id="user" name="username" value="{{@username}}" disabled
     > </td>
</tr>
<tr>
<td width="20%"> <label for="logtime">Logtime</label></td>
<td> <!--input type="text" id="time" name="alarmtime"-->
<input type="datetime-local" name="alarmtime" value="{{@alarmtime_corrected}}"
     disabled> </td>
</tr>
</table>
</td>
</tr>
</table>
</div>
<input type="hidden" id="note_id" name="note_id" value="{{@note_id}}">
<br>
<br>
<div align="right"><button type="button" class="w3-btn w3-white w3-border w3-
     border-grey w3-round" style="width: 150px" onclick="doCancel();">Cancel</
     button> <input type="submit" value="Submit"></div>
</form>
```

Acronyms

API Application Programming Interface. 1, 7

fMRI functional Magnetic Resonance Imaging. 2

HTML HyperText Markup Language. 1, 7, 8

NETE New Era Text Editor. 1, 3, 4, 7, 8

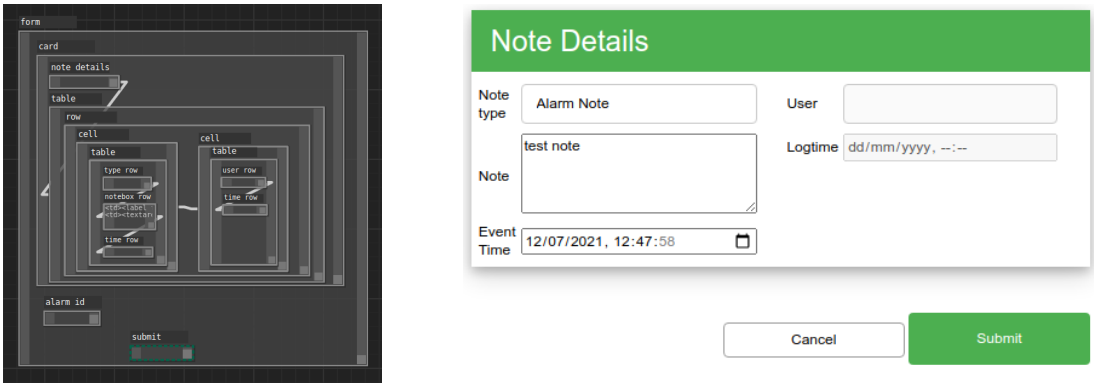
OHCO Ordered Hierarchy of Content Objects. 1, 3, 5

PHP PHP Hypertext Processor. 7

SMS Short Messaging Service. 1, 7

UML Universal Modelling Language. 2, 3

VPL Visual Programming Language. 1–3



(a) An html form example

(b) The real HTML page

Figure 12. HTML form example

References

[1] Garner and Stuart, "Reducing the cognitive load on novice programmers," *ED-MEDIA 2002 World Conference on Educational Multimedia, Hypermedia and Telecommunications*, 2002.

[2] M. Hansen, R. L. Goldstone, and A. Lumsdaine, "What makes code hard to understand?," *arXiv*, 2013.

[3] V. Damasiotis, P. Fitsilis, P. Considine, and J. Kane, "Analysis of software project complexity factors," 01 2017.

[4] C. S. Prat, T. M. Madhyastha, M. J. Mottarella, and C. Kuo, "Relating natural language aptitude to individual differences in learning programming languages," *Scientific Reports - Nature Research*, 2020.

[5] B. Shneiderman, "Cognitive psychology and programming language design," *SIGPLAN Notices*, 1975.

[6] M. Hansen, A. Lumsdaine, and R. Goldstone, "Cognitive architectures: A way forward for the psychology of programming," pp. 27–38, 10 2012.

[7] M. E. Hansen, A. Lumsdaine, and R. L. Goldstone, "Language design: A cognitive science approach (full presentation)," *Indiana University*, 2011.

[8] S. Markstrum, "Staking claims: A history of programming language design claims and evidence: A positional work in progress," in *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, (New York, NY, USA), Association for Computing Machinery, 2010.

[9] S. Markstrum, "Staking claims: A history of programming language design claims and evidence: A positional work in progress," *Evaluation and Usability of Programming Languages and Tools, PLATEAU'10*, 01 2010.

[10] A. A. Ivanova, S. Srikant, Y. Sueoka, H. H. Kean, R. Dhamala, U.-M. O'Reilly, M. U. Bers, and E. Fedorenko, "Comprehension of computer code relies primarily on domain-general executive brain regions," *bioRxiv*, 2020.

[11] E. E. Ogheneovo, "Software dysfunction: Why do software fail?," *Journal of Computer and Communications*, 2014.

[12] S. Janssens, U. P. Schultz, and V. Zaytsev, "Can some programming languages be considered harmful?," *PLATEAU'17*, 2017.

[13] D. E. Knuth, "Literate programming," *THE COMPUTER JOURNAL*, 1983.

[14] C. Simonyi, "The death of computer languages, the birth of intentional programming," *Microsoft Research*, 1995.

[15] N. Wirth, "A plea for lean software," *IEEE Computer*, 1995.

[16] G. Fairbanks and D. Garlan, *Just Enough Software Architecture: A Risk-Driven Approach*. 01 2010.

[17] K. Siau and P. Loo, "Identifying difficulties in learning uml," *Information Systems Management - Contemporary Practices in Systems Development*, 2006.

[18] M. PETRE and A. F. BLACKWELL, "Mental imagery in program design and visual programming," *International Journal of Human-Computer Studies*, vol. 51, no. 1, pp. 7–30, 1999.

[19] A. Von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[20] P. Dugerdil, "Assessing legacy software architecture with the autonomy ratio metric," *Software Engineering an International Journal (SEIJ)*, vol. 1, 09 2011.

[21] P. A. Fishwick, *An Introduction to Aesthetic Computing*. 2006.

[22] R. Navarro-Prieto and J. J. Cañas, "Are visual programming languages better? the role of imagery in program comprehension," *Int. J. Human-Computer Studies*, 2001.

[23] A. F. Blackwell, K. N. Whitley, J. Good, and M. Petre, "Cognitive factors in programming with diagrams," *Artificial Intelligence Review*, 1993.

[24] D. Te'eni and Z. Sani-Kuperberg, "Levels of abstraction in designs of human–computer interaction: The case of e-mail," *Computers in Human Behavior*, vol. 21, pp. 817–830, 09 2005.

[25] D. Gračanin, K. Matković, and M. Eltoweissy, "Software visualization," *Innovations in Systems and Software Engineering*, vol. 1, no. 2, pp. 221–230, 2005.

[26] L. Fang and D. Hepting, "Assessing end-user programming for a graphics development environment," vol. 6335, pp. 411–423, 08 2010.

[27] N. Valcasara, *Unreal Engine Game Development Blueprints*. Packt Publishing, 2015.

[28] C. Elliott, V. Vijayakumar, W. Zink, and R. Hansen, "National instruments labview: A programming environment for laboratory automation and measurement," *Journal of The Association for Laboratory Automation*, vol. 12, pp. 17–24, 02 2007.

[29] M. Lekić and G. Gardašević, "Iot sensor integration to node-red platform," in *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pp. 1–5, 2018.

[30] V. Yoghourdjian, D. Archambault, S. Diehl, T. Dwyer, K. Klein, H. C. Purchase, and H.-Y. Wu, "Exploring the limits of complexity: A survey of empirical studies on graph visualisation," *Visual Informatics*, vol. 2, no. 4, pp. 264–282, 2018.

[31] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, Nov. 2010.

[32] M. Kölling, *Using BlueJ to Introduce Programming*, pp. 98–115. 04 2008.

[33] T. Sharma, P. Mishra, and R. Tiwari, "Designite: a software design quality assessment tool," pp. 1–4, 05 2016.

[34] A. Telea, H. Byelas, and L. Voinea, "A framework for reverse engineering large c++ code bases," *Electr. Notes Theor. Comput. Sci.*, vol. 233, pp. 143–159, 03 2009.

[35] W. Bischofberger, J. Kühl, and S. Löffler, "Sotograph – a pragmatic approach to source code architecture conformance checking," vol. 3047, pp. 1–9, 05 2004.

[36] S. J. DeRose, D. G. Durand, E. Mylonas, and A. H. Rehear, "What is text, really?," *Journal of Computing in Higher Education, Vol. I (2), 3-26*, 1990.

[37] S. B. Fountain and K. E. Doyle, *Learning by Chunking*, pp. 1814–1817. Boston, MA: Springer US, 2012.

[38] M. Naim, M. Katkov, S. Recanatesi, and M. Tsodyks, "Emergence of hierarchical organization in memory for random material," *Nature Scientific Reports*, 2019.

[39] H.-J. Schulz, S. Hadlak, and H. Schumann, "The design space of implicit hierarchy visualization: A survey," *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, 2011.