Visual Programming

Matthew van Cittert

25 March 2009

1 Introduction

The study of visual programming was once an active field of study, but journal articles on the topic post 1990's are rare. Yet the idea of visual programming is no less valid. Graphical diagrams and modelling tools continue to be used to aid programming. Integrated development environments (IDEs) rely increasingly on non-textual graphics to improve the task of programming. Some tools, such as Netbeans, Blender and XMLSpy, go further to supply diagrammatic views of code. Without the limit of insufficient hardware, and with an increase in tools and libraries for the design of graphical user interfaces, a gradual shift has been made towards graphically oriented programming. This project proposes to investigate, devise and evaluate representations of programming, primarily graphical. This inlvolves investigation of the field of graphical programming and the theory behind the relation of text, graphics and human perception.

Visual programming has been primarily adopted in the audio and electronic fields, but not in the realm of general programming. Articles on visual programming have neglected study on why graphics are effective and what role graphics may play in the improvement of programming. To research abstractions for programming, it will first be necessary to establish a good understanding of the concept of text, and to investigate and establish the strengths and weaknesses of text and various graphical representations, such as diagrams, flowcharts and graphs. Part of this understanding may be available in the literature, the rest will need to be explored through tests and analysis. Abstractions may then be designed from this basis. These abstractions will be combined into demonstrational graphical programming systems. Finally these systems will be evaluated, to analyse whether the abstractions benefit programming. Analysis will focus on whether the use graphics would be feasible in programming systems, and defining the respective roles of text and graphics in programming.

2 Research problem

2.1 Research question

Current methods of mainstream programming rely largely on text, with few abstractions to facilitate the process and with little support from graphics. Our research question is:

> We aim to devise graphical and textual abstractions to facilitate the process of programming to improve the reliability, accuracy and design of programs and reduce the time and effort required to produce, alter, correct and understand code.

2.2 Clarification

To avoid ambiguity and misinterpretation, the research question requires concrete definitions of the concepts outlined. These are given below. Note that the terms are interrelated and partially overlapping. For example, to alter a program requires a good understanding of the code.

Reliability Frequency and severity of syntactic and semantic errors (bugs).

Accuracy Producing output that is as close as possible to that which the programmer intended.

Design Reducing code to reusable, simple to understand and logically grouped components which interact in clearly defined ways.

Production Generation of code that achieves the aim for which it was intended.

Alteration Change or extension of code to meet new specifications or expectations. This includes updating the rest of the system to be compatible with the change.

Examples

- Commenting out a variable and all further references to it.
- Changing a data structure from a queue to a stack and adjusting the system to become compatible with the change.
- Incorporating and reusing code in other projects with not completely compatible requirements.
- Updating, migrating or extending programs to serve new goals or run in different environments.

Correction

- Syntax: Find code that does not satisfy the constraints of the compiler, understand why it is not acceptable and rewrite the code so that it is acceptable to the compiler.
- Semantics: Find code that produces unintended results, understand why the code does not produce the expected results and rewrite the code such that it does what is expected of it.

Understanding Comprehension of what a program or segment of code does, how it goes about this and what the results, siginificance and consequences of the piece of code are.

2.2.1 Issues with current methods of programming

Control

- The style in which programs are written is entirely up to the programmer and may be difficult to understand and to follow.
- It is difficult, if not impossible, to place restrictions and exercise control over coding. It is entirely up to the programmer which features of a language to use, when and how.

Relevance to project

- Use of graphical or textual abstractions may be investigated to restrict construction of programs to easily readable layouts. This requires study into what constitutes an easily read layout. Some issues of readability such as lack of comments, abbreviated, uninformative or misleading variable names would be difficult or impossible to resolve. These issues require knowledge of the code and the logic behind it, which is available only to the programmer.
- Providing an interface between the user and code may provide a means to control the code produced.

Comments

- This is not a major issue as there are a number of formatters, or pretty printers, available to resolve poor coding style. Conversely, the difficulty of implementing automatic formatters in graphical programming systems is later raised as a difficulty in graphical abstraction.

Understanding

- Well written programs may be difficult (require a lot of time and effort) to understand. It is often difficult to determine from code what a program does without external documentation, which may not be supplied, as generation of documentation is often time consuming and tedious.

Relevance to project

- Graphical and textual abstractions may be used to improve navigation through code, abstract code to hierachies of constituent components and find representations that avoid excessive bloating of code from comments. Representations of code and documentation may be merged to avoid the need to use external documentation generators and modelling tools to hand craft external documentation.

Abstraction

- Few means are available to hide code not currently under study, such as code folding or placing the code in separate files.
- Different programming languages may use different words to express the same instructions.

Relevance to project

- Layers of abstraction may be used to show or hide details.
- Use of different views or representations may help focus and emphasise relevant points and features, while hiding others.
- Abstractions, especially graphical, may help to abstract and merge common features found in different programming languages.

Comments

- The adaptation of the Unified Modelling Language (UML) for use as a programming languages is suggested by [2].

Navigation

- It is difficult to navigate through code, requiring powerful IDE's, multiple files and search facilities.

Relevance to project

- Use of abstractions may aid navigation.
- The interface between user and code may be provided with search and navigation facilities.

2.2.2 Definitions and clarification of terms used

Easily readable An easily readable layout should require minimal effort to understand, follow and navigate. There should be defined starting and end points, and as few of these as possible. There should be defined and navigable paths through the content from start points to end points.

Well written

Function

- The program does what the programmer intended, in the simplest and most straight-forward way possible to achieve the efficiency required of the program.

Complexity

- Complex portions of code should be broken into simple components and combined so as to be easy to follow and understand.

Understanding

- Comments should be supplied where, and in such a way that, they improve understanding of the code. The layout should be easily readable and easy to follow, simple (complex algorithms broken into simple components, combined in easy to follow manner). Abbreviation should be avoided where abbreviated names take effort to resolve, but names should be as short as is feasible to avoid effort taken to read, remember and process them. But names should be memorable and suggest their purpose to foster understanding.

Simple Simple code is of a length and complexity that its entirety could be understood thoroughly and completely with little time and effort (where 'little' is a subjective term), pre-supposing that the components of the code are already understood in the same manner.

2.3 Outcomes of Project Research question

"We wish to devise and evaluate abstract representations of existing textual programming languages to improve the reliability, accuracy and design of programs, and to reduce the time and effort required to produce, correct, extend and understand them."

In view of the above topics and the research question, the goals of the project are to provide abstractions that achieve part or all of the following:

Understanding Represent code in forms that reduce the effort and time required to understand and interpret it, with less reliance on external documentation.

Aim Increased understanding of the code should result in improved program reliability and accuracy, and better understanding of the overall program. Better representation of the code may improve understanding of the problem, which may lead to improved program design. Reducing time and effort required to understand code would reduce the time and effort required to correct, extend or reuse the code.

Navigation Provide representations that allow readers to expend less time and effort, and result in less confusion in navigation of code.

Aim Better navigation of code should decrease the time and frustration involved in producing, extending and correcting programs. Representations emphasising logical code navigation may encourage better overall program structure and design, due to the benefit to navigation provided by logically grouping modular code.

Abstraction Provide means to reduce distractions while studying code. Enable the reader to focus on one aspect of a program and ignore the rest of code. Abstract specifics of language implementation.

Aim Removing distractions and encouraging focus on the code under study may reduce the concentration required to understand code. Abstracting common features of programming languages reduce the time and effort required when programming in different languages. Understanding by programmers whose native language is different to that used by a programming language may be improved. Producing programs in different languages should be easier and to an extent automated, as the programmer would not have to deal with the quirks of each language and the same abstraction could be applied to multiple languages.

Control Enforce a more readable style. Control which features of a language may be used, when and how.

Aim By improving the layout and presentation of code, the time and effort required to understand the code should be reduced. Otherwise obscure connotations and implications of the code should become clear. This should improve correction, alteration and extension of code. This may result in better program reliability. Readable code should be easier to navigate and should be easier to remember. Control of program features could be used to encourage good programming practice. This should result in more reliable code that is easier to follow, and to alter and extend.

3 Background

3.1 Introduction

Visual programming was once an active field of study. Despite this, many aspects of the theoretical basis of visual programming were neglected or understudied. A firm basis and understanding of the roles of text and graphics are needed before considering their appropriate uses in programming. Motivation for this study is warranted by the continued use of modelling tools for the creation of documentation, and the increasing reliance by IDEs upon graphical representations and decoration. With modern hardware and the availability of powerful software tools, visual programming no longer faces the limitations it did upon its inception, and may prove a valuable avenue of continued study.

3.2 Relevent concepts

3.2.1 Abstraction

Humans have a limited capacity to think. The overall number and complexity of ideas or concepts we can think simultaneously is limited. Abstraction allows a programmer to focus exclusively on relevant details of a piece of code, and ignore its inner workings. Functions may be used with no knowledge of how they achieve their task. Different graphical and textual representations, or views, of code will be investigated. This will aim to emphasise and give clarity to relevent features of code, and hide irrelevent features. There are many existing representations which may be drawn upon, such as graphs, diagrams, flowcharts.

3.2.2 Dimension

Within the scope of this project, dimension will be considered as a channel of information. Dimensions come in many forms. Some are visual, some audio, some tactile. This project will primarily consider visual dimensions, but audio may serve as an effective secondary channel of communication.

Visual dimensions include spatial dimensions, such as width, height and depth. Positional dimensions may take the form of coordinates. Colour may be divided into dimensions, such as shade, hue, red, green and blue. Audio includes dimensions such as pitch, volume or tone of voice. Tactile dimensions include texture and temperature.

Dimensions may be mapped to information. Scale of an object or position of a slider may be mapped to value. Colour and shape may be mapped to type. Position may be mapped to association, and order mapped to priority. This project will look at effective mapping of representation to information.

There are a number of considerations to take into account when evaluating whether dimensions are suitable as representations. The means to represent the dimension should be available. For example, touch and smell are not physically supported by computers. Despite this, a smell could be mapped to a description, such as lime or orange-blossom, or to a colour such as green or orange.

The dimension should have the necessary range to represent the number of values required of it. This range must be available at a granularity such that the values are adequately distinct. For example, the shades black and white could be used to accurately represent the boolean values true and false. The granularity could be reduced to allow 256 grayscale shades and so represent eight bit integers. But the shades would not be adequately distinct to accurately distinguish between shades. While this may be suitable for a rapid approximation, such as of a characters health level in a game, this representation would not be suitable for accurate arithmetic.

3.3 Project components

The project undertaking can be divided into three interrelated concepts:

3.3.1 Encoding

Text is an encoding. In its visual form, text is a way to read and write pictures. It has a defined set of graphics (letters) combined according to a defined set of rules to form words. Words are in turn combined, with more characters (punctuation) to form sentences. We are taught to read and write text. Text is a general purpose encoding, and has successfully been applied to programming, by deriving new rules and restrictions to create programming languages. Graphical programming could be viewed as tailoring a specialised form of reading and writing, namely reading and writing instructions to a computer. An important part of the project is to clearly define how graphics should be read and written. A complaint made by [5] is that graphics are considered a powerful form of communication due to the interpretation they facilitate, but that interpretation is not beneficial to a field requiring a precise and unambiguous notation. Rules are needed to govern how graphics should be arranged, what different configurations of graphics mean and how to interpret graphs, pictures and diagrams. This is essential to the successful inclusion of interpretation in a highly mathematical field. But provided the cost in time and effort required to learn the new encoding should not be prohibitively higher than its benefits, and the new encoding must be better suited to programming than those already in existence.

3.3.2 Interface

Producing graphical and textual abstractions implies adding a new layer between the user and the computer. This interface has roles of its own. An extra layer may simplify the process of programming by giving greater control over coding, automating code generation and abstracting details. But it may add costs. The translation of changes between layers is a minor performance cost on modern day hardware, but major programming undertaking, requiring diligence and often resulting in inconsistencies and other bugs. Without optimisation, abstraction of details causes the production of general purpose code. This code is not tailored to the task at hand and may be much less efficient than handcoded instructions.

3.3.3 Presentation

This refers to presenting code in a logical form. The closer the mapping from the problem to the solution, the easier the code will be to produce and understand [1]. This portion of the project consists of defining a basis of understanding of dimension, arrangement, association and interpretation, and looking at the way in which they may be used to improve the process of programming.

3.4 Text and graphics

3.4.1 Introduction

A distinction is commonly made between graphics and text (for example [4]). It is necessary to establish the relation between graphics and text to understand what is meant by graphical or textual programming.

3.4.2 Text and graphics

Text is a subset of graphics. It consists of a series of spatially two dimensional pictures, or characters, encoded serially to form a one dimensional string. But there is no term available to refer specifically to non-textual graphics. Where graphics and text are contrasted in this paper, the term graphics should be taken as means of representing information graphically other than by the use of conventional text. This includes the use of characters for decoration rather than function.

3.4.3 Graphical decoration of text

To make bare text easier to read, its graphical display is enriched or decorated. Following are examples of the use of presentation, without change of the underlying textual meaning, to provide visual cues to aid readers.

Character case Upper and lowercase letters provide two graphical forms of characters with the same interpretation. These may be used interchangeably while preserving the meaning of the word in the natural language. Capitalisation was used in case insensitive programming languages, such as Pascal, to emphasise, and so help the reader distinguish, keywords. Capitalisation is used in case sensitive Java to distinguish class names from identifiers. Camel case notation or underscores are used to help the reader distinguish separate words in a string of characters.

Text size A number of other alterations may be made to the display of text without altering the underlying meaning. An example is text size. Text size is not often used to decorate programming languages, but is used in text editors. Larger, more emphasised, text is considered more important and is commonly used in headings.

Font Different fonts are used to denote different classes of text. In the case of fonts, the characters have the same recognisable form but bear different styles of decoration. In many textbooks, and in many articles pertaining to computer science, different fonts are used to distinguish code samples from content.

Font Style Various font styles are employed in integrated development environments (IDE's) to distinguish parts of code. Styles such as bold, underlined, italic and strikethrough are used to separate keywords, identifiers, types, comments, directives and other parts of programming languages. These may also be used to highlight matching selected braces (Figure 2).

Shapes and pictures Another graphical emphasis is the use of shapes and pictures. In the Java Netbeans IDE (version 6.5), rectangles are drawn around certain items of text to provide emphasis, depending on the context. In text editors, a red wavy line underneath text often indicates a spelling mistake and a green one represents a grammar mistake. In some IDEs, wavy lines have been used to warn of syntax and semantic errors, such as unintialised or unused variables. Graphics may accompany text, such as yellow light bulbs to indicate suggestions, yellow triangles or red octogons containing exclamation marks to indicate warnings or errors, and blue or green circles containing an 'i' or a question mark to indicate information or help (Figures 2 & 3).

Colour Colour is widely used to emphasise and distinguish parts of text. In many IDES, such as Lazarus, Delphi, Netbeans, JCreator, CodeBlocks and VisualStudio, characters are given different colours to highlight keywords, symbols, identifiers, comments and directives, and to show that a stretch of text has been highlighted. The background colour of characters is used to highlight selected text, the current line, other instances of the selected word, or the prescence of a breakpoint. Colours may be used in the margin of a corresponding line to indicate the state of the text. Green is used to indicate that the line has been added this session and has been saved, orange or yellow to indicate that it has not been saved (Figures 2 & 3).

Layout Text is most commonly represented in a spatially two dimensional space, the screen. Special control characters, such as linefeed and carriage return, are interpreted as new lines. Lines and indentation is commonly referred to as whitespace. Whitespace is used to achieve layouts of text that improve readability. Conventions are devised and learnt to speed recognition and interpretation of the meaning assigned to different layouts. Some programming languages, such as Python, have incorporated layout into the language syntax.

Tables Columns and tables are structures that use the concept of layout. Tables have rules to aid interpretation. The first row of a table commonly comprises the table headings. These are commonly emphasised using styles such as bold or underlining. The items and headings in the same row or column are taken to be logically related (Figure 2).

Lists Bullets and points are structures used to aid layout and imply association. These lists may be ordered (often numbered) or unordered. The position of items in a list may have further implications. Items higher up in a list may be regarded as having higher priority or importance. This rule may be overriden in numbered lists. Items with a lower numbering are more important or have higher precedence (Figure 2).

Floating windows Floating text boxes and windows give meta-information about code and present options for auto-completion (Figure 2).

Summary This list is not exhaustive and there are many more applications of graphical decoration of text. Some of these aid the reader without affecting the underlying meaning, such as capitalisation in Pascal. Others allow the reader to make inferences about the contents, such as the priorities and associations in lists and tables. Others again determine the function, such as indentation in Python and capitalisation in case sensitive languages, such as Java. This list should emphasise the role of visual cues in aiding reading and interpretation of textual programming languages.

3.4.4 Textual abstractions and representations

A number of textual abstractions have been used in programming environments. Other abstractions have been used elsewhere, such as webpages and electronic documents, but have not been used in programming. It may be of worth to investigate, evaluate and possibly incorporate some of these unused abstractions into programming environments. It is also important to enumerate and evaluate textual abstractions, to determine whether their graphical counterparts are neccessary and superior, and to determine which graphical abstractions have no feasible textual counterparts. The results of these investigations may be used as a basis to justify or abandon the use of graphics in programming as a means of abstraction.

Abstractions of text used in programming

- Code folding, such as used for code hiding in the Visual studio, Lazarus and Netbeans IDEs.
- Code placed in separate files, used to organise data in operating systems.
- Hyperlinks, such as used for code navigation in the Netbeans, Delphi and Lazarus IDEs.
- Bookmarks, such as used for code navigation in the Lazarus IDE.
- Scrolling, allowing a workspace larger than the screen, found in many programs especially text editors.
- Functions, allowing lists of instructions to be abstracted to a single word, found in many programming languages.

- Line by line debugging and stepping through code, to relate program code and program execution.
- Graphical representation of code, such as in Blender, XML spy and Netbeans (Figure 4).

Abstractions of text not used in programming

- Overviews in the form of content pages and indices as forms of navigation, used in books and to navigate websites. Some IDEs, such as Netbeans, Visual studio, Delphi and Lazarus, show summaries of classes, methods and functions which could be viewed as use this abstraction.
- Tables or columns, as in spreadsheets and to format textual documents. This may be useful to represent parallel execution or multiple options side by side. This may include code for threads, if-then-else statements (as in Nassi-Scheider diagrams), case or switch statements and if-defined directives. But confusion may result if column divisions are not sufficiently clear, requiring more time and effort to read. Statements often fill the entire screen width. If columns are to offer an improvement to readability, statements should fit side by side readably.

3.4.5 Strengths and weaknesses of text

Advantages Text is a mature, tried and tested method of encoding and decoding information. It is well studied and well supported.

- Text exists in different modes, including visual, audio and tactile. Examples of these include writing, speech, braille, and derived encodings such as morse code. Communication of text is well supported.
- Programmers are already trained to read and write general text, as well as notations defined by programming languages.
- A large vocabularly exists and has been learned. This allows a single word to convey many complex concepts or descriptions. In these cases, words are very compact.
- Many facilities exist to support text. These include hardware, such as keyboards, and software, such as operating systems, text editors, special string structures built into programming languages and libraries of string handling functions.
- Mature, tested and well studied textual programming languages exist in abundance. These have tailored encodings of general text to suit programming.
- Graphical decorations of text have been devised, accepted and learned. Conventions, such as layout, have been established. Many IDEs effectively graphically augment text.

-
- Precise, mathematical etc. [5]
- Easier to recognise two words as distinct if lettering different (although sometimes read only first and last letter).
- Variation
- Compact
- Description - how to search for a picture?

Disadvantages

- Despite good support of communication of text in other modes, means of describing code informally is not always well defined. An example is the vocal description of a for-loop, such as "for(int i = 0; i <= 10; i++)", which is not described literally as "for open bracket int i ..." but rather "for i equals zero through ten" or "for i from zero to ten". In the same way, it is not unwarranted to expect that communication of encodings other than text may be defined or evolve.
- Text is general. Although text is specialised and tailored in the form of programming languages, text was not specifically designed to suit programming. There may be encodings that surpass text in its suitability for programming.
- Large and complex projects in textual programming languages rely heavily on external IDEs, documentation and modelling tools. In the case of layout, indentation may be part of the syntax, as in Python, or independant of it, as in Java, C and Pascal. In the same line of argument, it should be investigated whether representations to improve reading and writing code should be part of or derived from the syntax.
- Text takes time to read. Road signs reduce concepts to single icons. This supports the argument that words made up of multiple characters make it more difficult to a basic idea of the contents at a glance. But this abstraction is used to some extent in the form of functions - single, preferably descriptive, words representing lists of instructions. Applying the concept of road signs to programming may help readers gain an overview of code faster and with less effort than is currently required. But this relies on the use of memorable signs, in such numbers that they do not exceed the bounds of users' recollection and sufficiently disitinct not exceed the bounds of users' perception.
- Words are tied to specific languages. Images may have interpretive value. Road signs may be understood and easily recalled even if the reader does not understand the same language as the author of the sign, although

intpretation may be tied to cultural groups. This facility may play a role in the abstraction of specifics of languages' syntax, allowing common features to be abstracted to a common representation. But [5] raises the objection that programming requires an unambiguous, mathematical notation. In light of this argument, care should be taken to define the meaning of images to avoid ambiguity, and not to rely too heavily on interpretation.

3.4.6 Strengths and weaknesses of graphics

Despite the large amount attention the field of visual programming has received in the past, the use graphics may play in programming has not been given much attention [5]. A goal of the project is to investigate and demonstrate such uses.

Possible or claimed advantages of graphics

- Allows interpretation [4]. But text is also open to interpretation, as is the case with poetry. Interpretation allows the writer to invent new words which inherently communicate their meaning, such as scientific names composed of latin sub-words, or road signs. But the reliability of interpretation depends on the understanding, experience and grasp of the reader, and the reader's familiarity with techniques used by writer. [5] argue that interpretation is not be suitable in a precise, mathematical field such as programming as it is a cause of ambiguity.

- Expressive [3]. Provides more information with less clutter in less space [5]. This would depend on how the code is represented and is not inherent to graphics. The same claim could be made for different textual representations. The amount of clutter and irrelevant information presented depends purely on the amount of abstraction applied. Representations should aim for good use of space, not necessarily little use of space. [5] point out that the expressive power made available by visual cues, such as layout, are a side-effect and not part of the syntax. This leaves layout unregulated. The poor layouts that may result, especially in the case of inexperienced programmers, may cause confusion and reduce readability rather than improve it.

- Rapid and simple interaction and editing[3]. This may be true, but depends on the interface and is not inherent to graphics. Whether graphical interfaces are superior to text or consoles is a matter of debate and preference.

- Easy to learn [3]. Easier to understand [5]. This depends on the presentation complexity, memorability and mapping to the problem domain, and is not inherent to graphics.

- More efficient mental processing of graphics than text [4]. This is debatable. The representation is likely the most important factor in efficient mental processing, although use of visual cues such as colour are used in mind maps to make them easier to improve appeal and the ability to understand and remember them.

- Gives an overview, showing structure more clearly [5]. This may be true, but overview and structure are issues of abstraction, whether textual or graphical. But textual and graphical overviews should be compared to determine whether graphics could play a role in improving the quality of code overviews.

- Closer mapping to the problem domain [5]. This may be true as graphics is not faced with the restrictions of text. For example, blocks of information could be presented overlapping one another, although this has already been achieved with text boxes in IDEs such as Netbeans (Figure 2). But this would depend on the means used to represent the problem. This argument is supported by the prevalence of mind-maps and code modelling tools. The suitability of graphics and text need to be evaluated when mapping different parts of problems, such as structure, associations, interactions and relationships. The lack of textual restrictions may make it easier to apply anologies and metaphors to graphics.

- More memorable [5]. This may be true for some cases, for some people and with some pictures or words. The importance of memorable code and the relative ability to remember words, pictures and dimensions such as colour, should be investigated to determine the importance and role of this claim.

- Greater appeal[5]. [5] emphasise this as an important feature of graphics. It could be argued that a more appealing interface may lead to more enjoyable programming, resulting in more focus and less fatigue - as less effort is required to keep working than by an unwilling programmer. In the example of mind maps, decoration that has no function other than appeal is used to help unwilling students convince themselves to continue studying. Another avenue of investigation is the role of presentation and appeal in stimulating creativity. Included in establishing appeal are factors such as use of complementary colours and use of shapes.

Possible or claimed disadvantages

- In instances where images are used as code, the amount of memory required may be much greater than in the case of text. But this may not be the case where compressed or vector formats are used, nor should it be as much of an issue as it was in the past due to major improvements in hardware.

- The expressive freedom made available through the use of graphics may be misused, resulting in confusing and unreadable code [5]. This is an important point to keep in mind, but is only a problem if the syntax allows such abuses.

- Graphics are more difficult to read[5]. This depends entirely on the representation.
- Graphics are open to interpretation and so are not sufficiently precise and are open to ambiguity[5]. The same could be said for text, and depends entirely upon what contraints are built into the syntax.

3.4.7 Motivation, constraints, and research questions

If graphical representation has no place in programming then,

- Why are diagrams used?
- Why are mind maps and spider web diagrams used as study techniques?
- Why do modelling languages, such as UML, exist?
- Why is text graphically decorated?

If graphical representation is inherently superior to text [4] then,

- Why are diagrams not used exclusively?
- Why are there no general iconic programming languages or graphically extended languages in widespread use?

Needed research:

- What makes a good diagram, and what makes a bad one?
- What does it mean for a diagram to be good or bad?
- Under what circumstances and in what way are graphics more effective than text, or text more effective than graphics?
- What perceptions and conventions exists regarding representation?
- How may these perception and conventions be used or exploited to improve programming?
- What are the limits the optima and limits of human perception, memory and interpretation?
- What representations (diagrams, flow charts, graphs) exist, and what are their strengths and weaknesses?
- Is it better to use symbols users recognise, or train users to recognise symbols?
- What are the components of programming languages, and how are these most effectively represented under different circumstances?

- Are summaries or overviews useful to programmers? If so, then are textual or graphical representations more suitable as overviews? If so, then which representations and in which cases?

- Are textual or graphical representations preferable to represent structures, associations, interactions or relationships? If so, then which representations and in which cases?

3.5 Brief overview of available literature

3.5.1 Reviewed

General

- [1]: Evaluation of a programs facility. How closely the steps taken to code the solution to a problem map to the steps taken to solve the problem.

- [2]: Suggest that UML (Unified Modelling Language) could be used as a programming language, rather than just an external program modelling notation. But this would require extension of UML to provide more detail than it does at present.

- [5]: Graphics give the programmer more freedom of expression. A good understanding of the relation between the code and the problem is required to produce a readable solution. Counter-intuitively, graphical programming is not suited to beginner programmers, as poor code structure leads to code that is very difficult to read. Well defined notation is required to prevent ambiguity and misinterpretation of visual cues in graphical programming.

Graphical programming systems

- [4]: Presents Pict (Figure 5), an iconic programming language coded in Pascal. The authors completely reject text, claiming that icons alone should be used in progamming, and the user should never have to touch a keyboard. But this argument is not substantiated. Pict allows the user to graphical step through a program's execution. Pict allows a program to run, even if the code has not been completed. When the program reaches the end of available code, execution stops and a message is displayed. This may be very important for testing and design. Segments of code may be tested before continuing coding, without the use of stubs and dummy variables and functions.

- [3]: Presents Tinkertoy (Figure 6), a graphical interface for Lisp. Mentions the use of the graphical interface to ensure that programs are syntactically correct, as the interface will not allow icons to be placed in configurations that are not syntactically legal. Tinkertoy represents code using icons labelled with text. The user has instance access to icons that are not already displayed by typing the icon's name into a text box. The graphical icon is then created, with which the user can interact.

- [8]: Presents ThinkPad (Figure 7), a graphical interface for Prolog. Think Pad was coded in C. Structures are represented as graphical shapes with textual labels. Details, such as constraints ("==", "<", ">" "<=", ">=", "!="), are represented with text. ThinkPad allows the user to choose a shape - a rectangle or circle - for each structure they wish to represent. This implies a lack uniformity and structure in the graphical notation.
- [7]: Presents GARDEN (Figure 8), a graphical programming language. The authors of GARDEN claim that with GARDEN, the graphical representation is the code. This would imply that execution is determined by evaluation of the pixels of the image, which does not appear to be the case. The view of GARDEN's authors is that good graphical representation of code requires different views of the code to suit different purposes. For example, sometimes a graph or flowchart may be appropriate, whereas at other times a diagram would better represent the structures and processes.

3.5.2 Further promising literature

General

- "Parsing Visual Languages with Picture Layout Grammars": Defining the syntax of visual languages.
- "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework": Evaluation of visual languages.
- "Translucent Patches": Use of transparency as a dimension of communication. Form of dynamic representation e.g. a blue variable on a yellow function becomes a green variable.
- "The Hyperbolic Browser: A Focus + Context Technique for Visualizing Large Hierarchies": Use of focus points to expand detail in areas of interest, and reduce detail in areas not under study.
- "Learning to Read Graphics: Some evidence that "seeing" an Information Display is an Acquired Skill": Journal article on which [5] is based.
- "Visual Programming Languages: A Perspective and Dimensional Analysis": Classifications of visual programming languages.

Graphical programming systems [4]

- Pecan and Magpie: Systems which provide graphical extensions to pascal.
- Mira-3D: Use of three dimensional graphics in programming.

- Programming by Rehearsal: Uses the analogy of a theatre to facilitate programming, as an “operational metaphor”. Programs are presented as productions. Different parts of the program are referred to as performers, cues, stages, wings and troupes. The use of analogies in this way is one of the aspects to be looked at in the project.

3.6 Summary

Abstraction is a powerful but under-used concept in the realm of textual programming. Further inclusion of non-textual graphics in programming could be used to harness this concept. Dimension is another important concept. An understanding and investigation of dimension and our perceptions is important before attempting to map visual and auditory cues to information.

The project will consist of different portions addressing different issues. An investigation of encoding will be required to devise means for new notations to be read and written, in sufficiently precise and unambiguous notation to suit programming. The role of the interface will be to allow control of code and assist with navigation. The presentation of code refers to factors such as mapping the problem at hand to the code written to solve it, and use of abstaction and representations to focus on relevant parts of the problem.

Text is a powerful notation, but is better suited to programming with the aid of non-textual graphics and decoration. Precisely when text or different graphical representations surpass one another as means of representation will need to be established before deciding which representation to use when.

A number of articles have been written, raising a number of interesting points and outlining the design of a number of interesting systems. But the basis of the roles of text and graphics is understudied.

4 Design

4.1 Mapping to the problem domain

The visual cues available will need to be enumerated, and effective mappings of cues to information will need to be devised. A preliminary list of dimensions and associations are given in Appendix B on page 30. A number of considerations should be taken into account:

Coherence The concepts of coherence and consistency should be important factors in improving readability. A number of options exist, which should be evaluated. For example, all data could be mapped to a colour, and different data structures such as primitives, records and objects then distinguished by shape. Or the mapping could be reversed, distinguishing by shape, then colour. Different types, such as integers, characters and booleans, could then be distinguished by further detail or decoration, such as numbers or colours of dots. Alternatively all data structures could be given a common shape and colour, and then further distinguished by decoration. It should be investigated whether there are any significant differences in the user responses to the various design options, to justify the final approach chosen.

Complexity The limits and optima of user perception, memory and understanding should be established if a system is to be designed on more substantial grounds than experience or sheer luck. It will be necessary to determine how much to display and how complex the content should be, ultimately how busy the screen may be without user becoming overwhelmed. The representations used will then need to convey the required information while limiting detail to within these bounds.

These bounds may change dramatically with exposure and experience. As it is unlikely that any system will be complete and in use long enough to measure response of experienced users, responses to related systems could instead be measured. Example are strategy games and modelling languages. Both consist of a number of symbols which interact in defined ways. Establishing the on screen complexity and the ability of experienced users to differentiate and identify these symbols and their roles may give a sufficient idea of the constraints the system will face.

Some of the complexity considerations include surface structure of shapes, complexity of shape combinations, detail of decoration and colour complexity.

4.2 Considerations of project components

A number of questions regarding the different project components should be investigated to guide and justify design decisions.

Encoding

- Form combinations from a set of 'letters', or use a new character for every 'word'?
- How will start and end points be defined, if the encoding is not serial (such as two or three dimensional)? Structured programming prescribes the use of control structures with one entry and one exit point [6], precluding the use of goto statements. To take advantage of extra spatial dimensions, is there a way to define multiple entry and exit points while maintaining readability and structure? Is there any advantage to multiple entry and exit points?
- What definitions and contraints are required to ensure good structure and readability? Should the goal be more detail, at the cost of adding complexity, or greater simplicity, at the cost of control and precision?
- Should the project focus on graphical extensions of existing languages, or investigation of purely visual programming languages?

- An argument raised by [5] is that the power of visual programming languages lies in the range of visual cues made available. Yet new freedoms, such as removal of textual layout restrictions, are often side effects and not part of the language. The effective use or abuse of visual cues is then left entirely to the programmer. Should layout be part only of the interface, such as found in Java, C++ and Pascal, or part of the encoding, such as in Python? Should rapid but messy debugging, and expressive freedom, be traded for enforced good structure and readability?

Interface

- Structured programming.
- Abstraction of implementation specifics.
- Debugging.
- No syntax errors (Correctness).
- Navigation (layers of abstraction).
- Combine text and graphics.
- Control.
- Features: If-defs, commenting out variables, looking glass.

Presentation

- How should data structures and algorithms be presented? When should text, diagrams or graphs be used, and which diagrams or graphs? Should representations be restricted purely to one existing system, such as UML, or should different representations be used where appropriate?
- A comment made by [7] is that computer languages are made up of sub-languages, such as expressions and control structures. If languages are to be viewed in this way, how should each component of a language be represented under different contexts?
- An important decision is the mapping of visual cues to information. What associations already exist? For mappings provided as decoration by the interface, should the user have choices, such as colour and font options offered by many IDEs, or should the mappings be set?
- Static representation refers to a series of steps, which may be viewed individually in a sequence (Figure 1a). If the representation of the process changes after application of each step (Figure 1b), this will be referred to as dynamic representation. What are the benefits and disadvantages of representing processes statically or dynamically?

- Levels of abstraction.
- Analogies.
- Use of textual or graphical summaries (eg table of contents).

4.3 Pre-design tests

- Memory
  - How many shapes before "swamping" effect?
  - Strategy games - number and complexity of units.
- Perception
  - How complex shapes before difficult to distinguish?
  - Granularity of colours before difficult to distinguish.
- Interpretation
  - Limits of learning.
  - Is an overview helpful?
  - Is the devised graphical overview more suitable than the devised textual overview?
- Appeal
  - Complementary colours.
- Other
  - Types of languages and possible contributions/effectiveness by graphics.

4.4 Rerepresentation of existing languages

4.4.1 Goals

- Reduce time taken to code.
- Give control over code, to enforce good programming practices.
- Improve code navigation.
- Improve code structure.
- Aid debugging.
- Abstract implementation details.
- Make code easier to understand.

4.4.2 Evaluation of system

- Reduce time taken to code: Compare times taken to produce equivalent code using the graphical system, and using text with an appropriate IDE. Considerations:
  - Expertise using graphical system.
  - Expertise using text.
  - Expertise using IDE.
  - Ordering of tests.
  - Requirements, such as number of replicates, required for statistical analysis.
- Give control over code, to enforce good programming practices: Calculate number of syntax errors made during coding using text with an appropriate IDE (syntax errors should be impossible with graphical interface). Calculate the time spent correcting syntax errors. Calculate what portion of time is saved by omition of syntax errors, and whether this is statistically significant. Test may not be feasible. Considerations:
  - Expertise using text.
  - Expertise using IDE.
  - Requirements, such as number of replicates, required for statistical analysis.
  - Means to count errors and time correction of errors. Manual counting and timing may not be accurate enough.
- Improve code navigation: Compare times taken to navigate about a program to insert missing code, or correct a set of errors. Considerations:
  - Should be based on problems encountered in real programs.
  - Expertise using graphical system.
  - Expertise using text.
  - Expertise using IDE.
  - Requirements, such as number of replicates, required for statistical analysis.
- Improve code structure: Subjective evaluation of structure of textual code output from a textual program and from the graphical system. Or use a contrived example where good structure is required to make the code function. Considerations:
  - Subjective preferences.
  - Programming experience.

  - Exposure to graphical modelling tools.
  - Analysis of subjective data.

- Aid debugging:
- Abstract implementation details:
- Make code easier to understand:

4.4.3 Rough ideas for system

- Reduce time taken to code.
  - Expose methods of wrapped classes at the click of a mouse.
  - When a piece of code is commented out, comment out all dependent parts of code.
- Give control over code, to enforce good programming practices.
  - Only allow syntactically correct expressions.
  - Give the ability to enable or disable icons, representing statements of programming language.
- Improve code navigation.
  - Use navigation techniques designed for rapid navigation around strategy games.
  - This includes using a workspace that is larger than the screen.
  - To navigate about the workspace, the user can pan, zoom, or use a minimap (a simplified overview of the entire workspace).
  - There may also be labelled arrows on request, to give directions to offscreen code.
  - There may also be control groups of code assigned to hotkeys. The user may cycle through blocks of code within a control group.
  - Allow user to follow associations between parts of code.
- Improve code structure.
  - Allow the programmer to group logically related data structures and algorithms. Near and far could be mapped to degree of relation. This could use thresholds for near and far or use cladistics to create subgroups. But a simpler option would be to create borders and map within and without to related and unrelated.
- Aid debugging.

  - Possibly place checkpoints in code to allow logging of program execution. Use logged information to allow graphical stepping through code that has been executed. Stepping through code in the process of execution and use of break points may integration with a debugger. This would most probably be a major undertaking, requiring advanced knowledge of the inner workings of the debugger. It would be better left as an extension to the project.

- Abstract implementation details.
  - Represent common features of programming languages using the same icons.
  - Allow the user to restrict icons to a subset common to specified languages. This will allow the same graphical blueprint to generate code for multiple languages.

- Make code easier to understand.
  - Allow user to expand or abstract regions of code.
  - Use focus points to show the details of regions of code near the point, while gradually abstracting code further away.
  - Indicate associations between parts of code.
  - Use uniform and consistent visual cues (colour, shape, patterns) to indicate logical relationships.

4.4.4 Diagrams

4.5 New representations

4.5.1 Goals

- Devise forms of programming that give a closer mapping of problem and solution.
- Use appeal value of graphics.

4.5.2 Evaluation of system

- Mapping
  - Program walkthrough [1].
- Appeal
  - Opinion of users.

4.5.3 Rough ideas for systems

- Static and Dynamic
- Strategy game (beans) - overview only
- Threads (workers)

4.5.4 Diagrams

5 Implementation

5.1 Coding

5.1.1 Tools

- Pascal
- Lazarus
- FPC

5.1.2 Advantages

- Open source
- GUI development
- Private codebase
- Cross platform
- Personal Experience

5.1.3 Disadvantages

- Codebase of C and C++ very large.
- Java and C# considered "safer".

6 Summary

- Graphics are not inherently better and the design of the system will decide whether the use of graphics improves or hinders code production and readability. Graphics may provide many abstractions to aid programming, but many of these abstractions may also be implemented in text. The benefits of augmenting texts with graphics will need to be tested to determine whether graphical programming is a feasible avenue for further study, how it may benefit the field of computer science, and when, where, how and in what way its use is appropriate and beneficial.

References

[1] Brigham Bell, John Rieman, and Clayton Lewis. Usability testing of a graphical programming system: things we missed in a programming walkthrough. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 7–12, New York, NY, USA, 1991. ACM.

[2] Morgan Björkander. Graphical programming using uml and sdl. *Computer*, 33(12):30–35, 2000.

[3] M. Edel. The tinkertoy graphical programming environment. *IEEE Transactions on Software Engineering*, 14(8):1110–1115, 1988.

[4] E.P. Glinert and S.L. Tanimoto. Pict: An interactive graphical programming environment. *Computer*, 17(11):7–25, 1984.

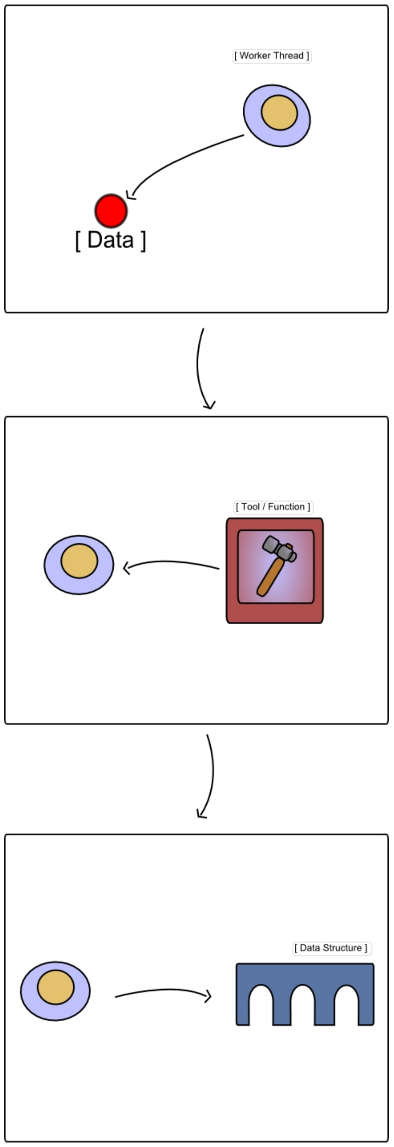
[5] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[6] Edwin D. Reilly and Francis D. Federighi. *Pascalgorithms*. Houghton Mifflin Company, Boston, 1989.

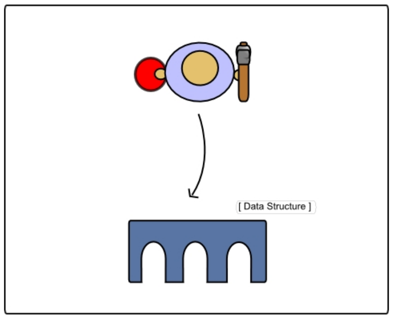
[7] Steven P. Reiss. An object-oriented framework for graphical programming (summary paper). In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 49–57, New York, NY, USA, 1986. ACM.

[8] R.V. Rubin, E.J. Colin, and S.P. Reiss. Think pad: A graphical system for programming by demonstration. *IEEE Software*, 2(2):73–79, 1985.

Appendix A: Diagrams



(a) An example of a static representation. Each step in the process is presented individually. The process is represented as a sequence of steps, executing from top to bottom.



(b) The same process represented in (1a), but represented dynamically. Rather than showing each step, the representation has changed to indicate the state of the process, allowing the reader to infer which instructions have been given until this point.

Figure 1: An example of static (1a) and dynamic (1b) representations of a non-existant graphical programming language.

Appendix B: Preliminary list of dimensions

Examples of dimensions

- Visual
  - Spatial: width, height, depth, length, volume (size).
  - Positional: coordinates, rotation.
  - Structural: shape, detail, outline, completeness.
  - Surface: colour, brightness, transparency, contrast, gradient, blur, reflectivity, shade.
  - Other: number, density, arrangement, completeness.
- Tactile
  - Temperature, texture.
- Audio
  - Volume, pitch, frequency, origin.
- Patterns of any of the above. If the above are viewed characters, then patterns may be viewed as words.
  - Checkerboard, stripes, music.

Dimensions may have values or thresholds. For example, colour may be blue, size big, temperature cold, texture rough or volume loud.

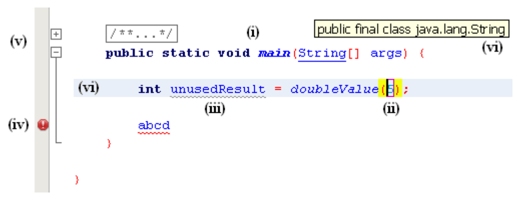
Examples of existing associations

Positional coordinates

- Down or right indicates more progress and left or up indicates less, in the case of sliders, scrollbars and progress bars.
- Relative position may indicate precedence, where items above have higher priority than items below.
- Items grouped together are interpreted as associated.

Rotation

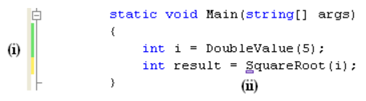
- Rotation may be used to represent the viewers virtual orientation, such as in a flight simulator.



public final class java.lang.String

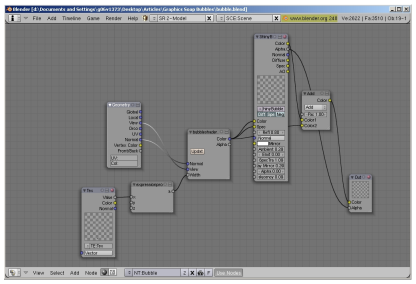
```
/**...*/
public static void main(String[] args) {

    int unusedResult = doubleValue(5);

    abcd
}

}
```

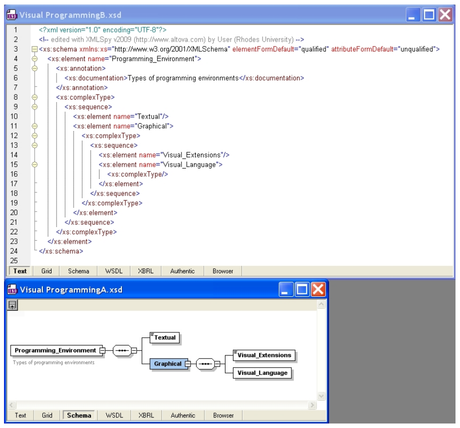
Figure 2: Use of graphical decoration by Netbeans version 6.5. (i) Fontstyles and colours used to emphasise and distinguish keywords, identifiers and symbols. “String” underlined to indicate a hyperlink when Ctrl held down and mouse moved over. Rectangle drawn around comments to indicate folded code. (ii) Matching braces at mouse position highlighted with yellow. Number five surrounded by a red rectangle to emphasise newly typed parameter. “doubleValue” in italics to emphasise edited function. (iii) “unusedResult” underlined by a grey wavy line (green when line not highlighted) to indicate that it is never used and warn that the line is superfluous. (iv) Red octogon containing an exclamation mark to draw attention to synax error. “abcd” underlined by a red wavy line to indicate that it is syntactically incorrect. (vi) Currently selected line highlighted. (v) Squares containing “+” or “-” to control and indicate whether code is folded. (vi) Floating text box to give meta-information about code.



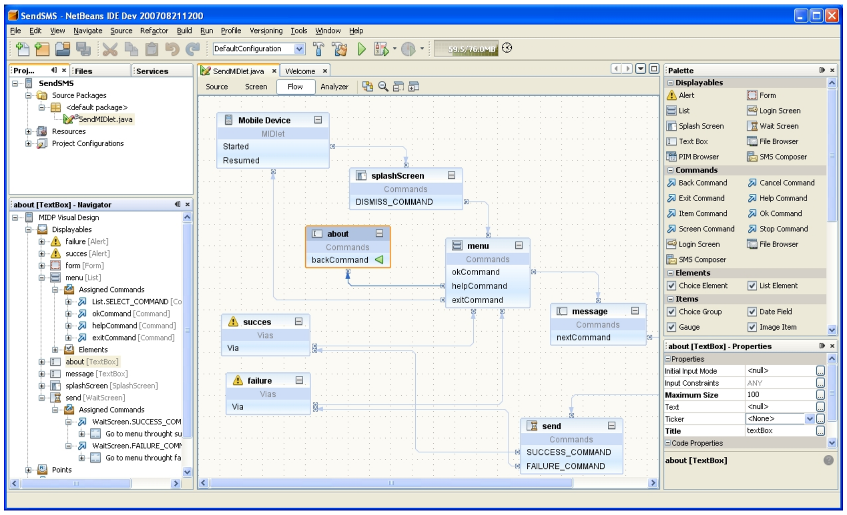
```
static void Main(string[] args)
{
    int i = DoubleValue(5);
    int result = SquareRoot(i);
}
```

Figure 3: Use of graphical decoration used by Visual Studio 2008 Express Edition. (i) Margin colours used to represent code added in the current session. Green indicates that it has been saved, orange that it has not. (ii) The square under the ’S’ of “SquareRoot” indicates extra options the IDE may perform on the SquareRoot function. In this case the dialog expands when clicked on to allow the IDE to auto-generate a stub function.


(a) Visual editing of a Python script in Blender (version 2.48).


```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2009 (http://www.altova.com) by User (Rhodes University) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="Programming_Environment">
    <xs:annotation>
      <xs:documentation>Types of programming environments</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Textual"/>
        <xs:element name="Graphical">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Visual_Extensions"/>
              <xs:element name="Visual_Language">
                <xs:complexType/>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(b) Visual editing of an XML schema in Altova XMLSpy (2009 Enterprise Edition).

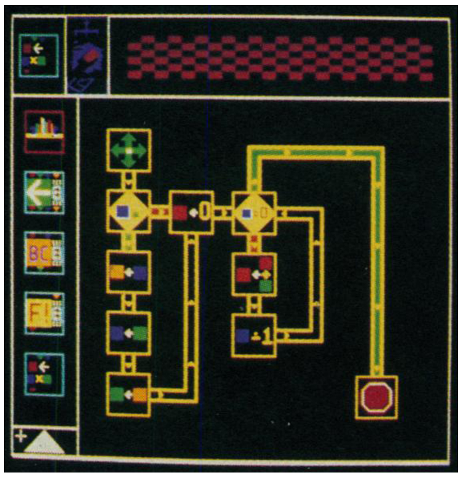


(c) Visual editing using the Visual Library of the Java Netbeans IDE version 6.5 (from http://graph.netbeans.org/screenshots/mobility60.png).

Figure 4: Examples of visual editing implemented in existing software packages. To gain an understanding of the possible roles text may play in programming, it is important to investigate where, how and why graphical editing is used. The above examples include visual editing of a Python shader (4a), an XMLSchema (4b) and a Java program (4c).

```
type
  SixDigits = 0 .. 999999;

function Red( Green, Blue: SixDigits ): SixDigits;
(* multiply two natural numbers by means of repeated addition *)
var
  Orange: SixDigits;
begin

  if Blue > Green then
  begin
    Orange := Blue  ;
    Blue   := Green ;
    Green  := Orange;
  end;

  Red := 0;

  while Blue > 0 do
  begin
    Red  := Red + Green;
    Blue := Blue - 1;
  end;

end;
```

(a)



(b)

Figure 5: Blue > Green: Big blue square, small green square. In a triangle representing the conventional if statement. Swap red, orange, green: arrows. Green for true, red for false. [...] Power in use of symbolism. But symbols are not part of language. Programmer could give any image as icon, or no image at all & still syntactically correct. Some concepts, esp. abstract, may not be possible to draw as simple icon (eg. fib written on icon). Require artist skill - programming = mathematical, cannot expect all programmers to be good at art (quite the opposite). Time to draw. Given icons easy to read once know what they represent. But may be difficult to intepret. Ambiguity. Perhaps easier with experience. Use conventions (triangle, arrows) = hints at formal language, but not used. Too much unconstrained freedom. Finding function may take time (no way to search as no names, need search for similar pictures algorithm, + take time to draw search template). Will difficult to distinguish if many colours (variables) involved.

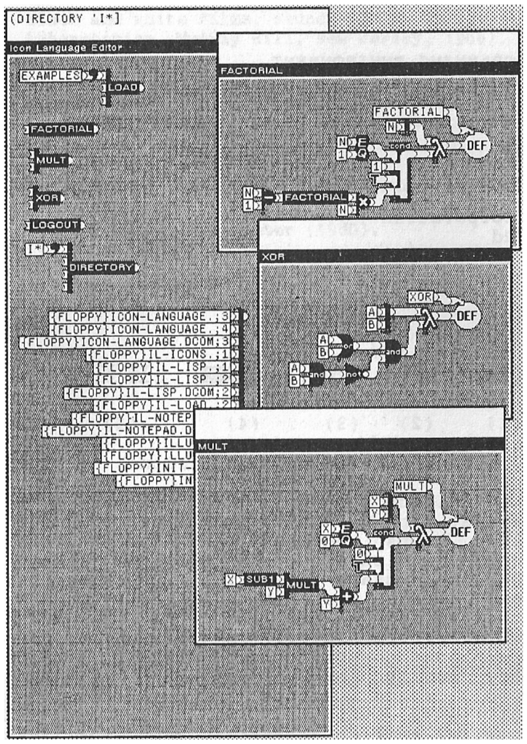


Figure 6: Tinkertoy[3] is a visual front-end for the Lisp programming language. Icons, representing data and algorithms, are combined together to form programs. Correct syntax is ensured as icons will only combine if the combination is syntactically correct. As Lisp is an interpreted language, the functions are evaluated immediately. The output is represented as an icon with which the user may in turn interact and produce further structures. Tinkertoy gives instant access to icons not already displayed on the screen by allowing the user to type the name of the atom or function in a text bar.

Colour

- Blue and cold, and red and hot, are associated.
- Red and danger are associated.
- Yellow and black are used as a warning colours.
- Blue and green are used for information.

Transparency

- Transparency may represent prescence, contribution or weight.
- A faded music control would be expected to make less noise.
- Transparency may indicate an inactive or infrequently used control.
- A dragged window or a control blueprint that has not yet been placed is often represented as transparent.

Shade

- Light is associated with good, dark is associated with bad or evil.
- Grey is used to represent disabled.

Size

- Large volume is associated with high value.

Blur

- Blue is associated with speed. A fast movement may be represented statically by blurring.

Volume

- A high volume is associated with a high value.
- A loud noise may be interpreted nearby, a muffled or soft noise as distant.

Pitch

- A high pitch is associated with warning, such as an alarm bell.

Frequency

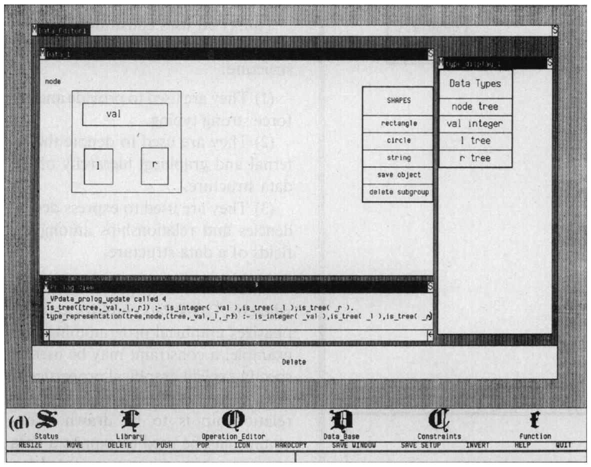
- A high frequency is associated with nearby, such as a metal detector.

Temperature

- Hot and warm are associated with nearby, cold with far away.

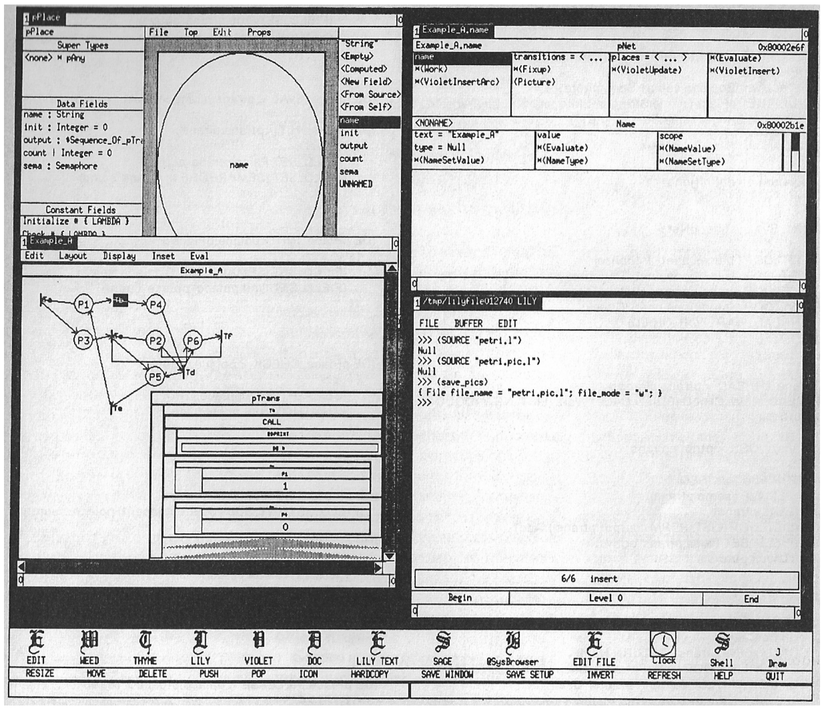
Texture

- Rough or unpolished are associated with unfinished.



```
_VPdata_prolog_update called 4
is_tree([tree,_val,_l,_r]) :- is_integer( _val ),is_tree( _l );is_tree( _r ).
type_representation(tree,node,[tree,_val,_l,_r]) :- is_integer( _val ),is_tree( _l ),is_tree( _r
```

Figure 7: ThinkPad[8] is a visual front-end for the Prolog programming language. The example in the figure is of a tree node containing an integer as data. ThinkPad allows the user to choose either a rectangle or a circle to represent a structure. This may lead to confusion if there are style inconsistencies between programs or programmers.


pPlace
File Top Edit Props
Super Types
<none> ⌘ pAny
Data Fields
Constant Fields
Example_A
Edit Layout Display Inset Eval
P1 P2 P3 P4 P5 P6
Tf Td
pTrans
CALL
Example_A.name
pNet
<NONAME>
Name
FILE BUFFER EDIT
6/6 insert
Begin Level 0 End
EDIT WEED THYME LILY VIOLET DOC LILY TEXT SAGE @SysBrowser EDIT FILE Clock Shell Draw
RESIZE MOVE DELETE PUSH POP ICON HARDCOPY SAVE WINDOW SAVE SETUP INVERT REFRESH HELP QUIT

```
name : String
init : Integer = 0
output : $Sequence_Of_pTra
count | Integer = 0
sema : Semaphore
Initialize # { LAMBDA }
```

```
Example_A.name                    pNet                         0x80002e6f
name                  transitions = < ...   places = < ... >    ⌘(Evaluate)
⌘(Work)               ⌘(Fixup)              ⌘(VioletUpdate)     ⌘(VioletInsert)
⌘(VioletInsertArc)    (Picture)

<NONAME>                          Name                         0x80002b1e
text = "Example_A"    value                 scope
type = Null           ⌘(Evaluate)           ⌘(NameValue)
⌘(NameSetValue)       ⌘(NameType)           ⌘(NameSetType)
```

```
/tmp/lilyfile012740 LILY
>>> (SOURCE "petri.l")
Null
>>> (SOURCE "petri.pic.l")
Null
>>> (save_pics)
{ File file_name = "petri.pic.l"; file_mode = "w"; }
>>>
```

Figure 8: GARDEN[7] is a visual programming language. GARDEN provides multiple views of a program, such as petri nets, finite state automata and Nassi-Schneider diagrams. The aim of GARDEN is to provide a language where the diagram is the code, and does not rely on an underlying textual language. This approach intends to avoid the serial one dimensional restrictions of textual languages.