Redesigning the IDE

Submitted in partial fulfilment of the
requirements of the Bachelor of Science
Honours degree in Computer Science
Rhodes University

Matthew van Cittert

10th November 2009

Acknowledgements go to my supervisor, Shaun Bangay, the members of my project workgroup (VRSIG) and the Computer Science Department of Rhodes University.

I also acknowledge the financial and technical support of this project of Telkom SA, Comverse SA, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University.

Chapter 1

Abstract

The design of many existing Integrated development environments (IDEs) has been based on a modified text editor. This project looked at redesigning this interface, to make code easier to write, navigate and understand. Predictive and heuristic evaluations support this hypothesis, but the current design requires the programmer to adopt a specific style.

Chapter 2

Research Question

2.1 Problem Statement

2.1.1 Existing IDEs and the lack of change

Integrated development environments (IDEs) are important tools. Yet despite the rich feature set and ongoing development of IDEs, their underlying design has not changed from that of a text editor. This project redesigns the IDE to provide an interactive representation of code that is more suited to programming than the enhanced text editor currently offered by IDEs. The aim of this interface is to provide mechanisms and representations that make it possible to more efficiently write, navigate and understand object-oriented code than is possible with any existing interface.

2.1.2 Problem statement definitions

To clarify what is meant by the three core aims of the project (writing, navigating and understanding), their definitions as used in the project are given below.

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

This includes the change or extension of code to meet new specifications or expectations, and updating the rest of the system to be compatible with the change. In the context of this project, this means creating a class or a class member, such as a field.

Navigate Find existing code and display it in the visible work area of the screen. In the context of this project, navigation means showing the code for a class on the screen in the case of a traditional IDE, or showing a class GUI on the screen in the case of the project interface.

Understand Comprehension of what a program or segment of code does, how it goes about achieving this goal, and what the results, significance and consequences of the piece of code are, by anyone who wishes to understand how the code functions. In the context of this project, this means how classes are related - which classes are derived from a common ancestor class, and which classes contain instances of which other classes as fields.

2.1.3 The need for change

Text is a good medium for expressing code. It is compact, easy to read and quick to type. But the text editors used by many IDEs impose restrictions and hindrances upon the user.

- The spread of code across numerous files requires the user to jump between pages of code.
- Bulky comments and error handling code, which are irrelevant to solving the problem, are interspersed among lines of code that do contribute to the solution. These comments and error handling blocks are important to the understanding, debugging and operation of the code, but obscure the underlying algorithm.

- Code is viewed at a single level of detail.

Effective workarounds have been developed for many such issues.

- Summaries of the classes and methods in a project give information and allow navigation among many files, as though they were a single file.
- Tools, such as FPDoc, allow the user to integrate comments from separate files and display them as hints while viewing the associated code.
- The uniform level of code detail is offset by programming language design, such as reducing a list of instructions into a single function identifier, or by innovations such as code folding.

But these IDE extensions are not always enough, or otherwise have avenues for improvement. This project looks at the way code is represented, with the aim of lessening these issues by changing the format in which code is presented, rather than only adding increasingly complex features to the text editors traditionally used.

2.1.4 Existing solutions

Existing text-based IDEs commonly offer a number of services. To make code easier to write they may offer

- code generation from templates,
- code auto-completion,
- and support for code refactoring.

To make code easier to understand, they may offer

- tools to generate formatted documents from comments,

- highlight parts of syntax, syntax errors or potential semantic errors (such as uninitialised variables),
- and provide summaries of types and routines accompanied by iconic aids to give extra information, such as visibility.

To make code easier to navigate, they may

- provide means to show references to an identifier,
- provide different views and summaries of types, routines and files,
- provide hyperlinks, bookmark facilities and shortcut keys to jump between definitions and declarations,
- provide search and replace facilities,
- and allow operations to take place over multiple files, simulating one large file.

To make code easier to read, text editors commonly graphically decorate text.

- In the past, capitalisation was commonly used to differentiate keywords from identifiers.
- Font style (bold, italics, underlined), background colour and foreground colour are used for the same purpose, and to emphasise breakpoints or the currently selected line (Figure 2.1 on page 8, Figure 2.2 on page 8).
- Icons, boxes and coloured lines are used to show warnings, information, highlight syntax errors and show which lines of code have been edited since the last save (Figure 2.1 on page 8, Figure 2.2 on page 8).
- Whitespace is used to achieve a two dimensional layout of otherwise serial text. Usually this is just for readability, but in languages such as Python it has been included as part of the syntax.

- Although not yet commonly (if at all) included in programming text editors, differences in font type and size are commonly used to different parts of text - such as ordinary information from programming language code. General text editors also commonly use columns, tables and lists to spatially group or separate parts of text.

- Floating windows are used to give hints and meta-information about code, such as comments. These windows may also be used to give auto-completion options.

These techniques are still effective when editing individual methods, as has been shown by the number of unsuccessful attempts to edit methods using other representations. And added to the above points, visual programming has again begun to make a reappearance in modern IDEs.

- Microsoft Visual Studio includes graphical representations of the classes in a project, known as a class diagram view (Figure 2.3 on page 9).

- The 3D modelling tool Blender also provides graphical representations of Python scripts (Figure 2.4 on page 10).

- A number of UML-based modelling tools, such as Rational Rose, are capable of generating code from UML diagrams.

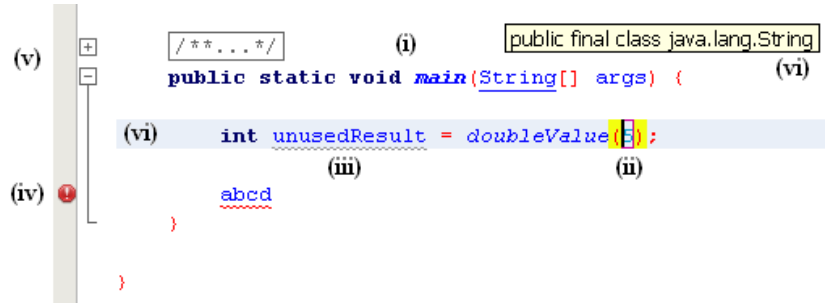


Figure 2.1: Use of graphical decoration by Netbeans version 6.5. (i) Font styles and colours used to emphasise and distinguish keywords, identifiers and symbols. "String" underlined to indicate a hyperlink when Ctrl held down and mouse moved over. Rectangle drawn around comments to indicate folded code. (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.

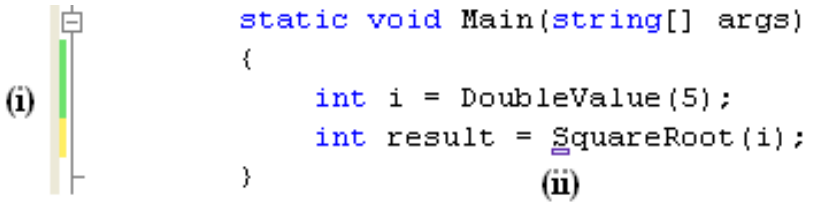


Figure 2.2: Use of graphical decoration used by Visual Studio 2008 Express Edition. (i) Margin colours used to represent code added in the current session. Green indicates that it has been saved, orange that it has not. (ii) The square under the 'S' of "SquareRoot" indicates extra options the IDE may perform on the SquareRoot function. In this case the dialog expands when clicked on to allow the IDE to auto-generate a stub function.

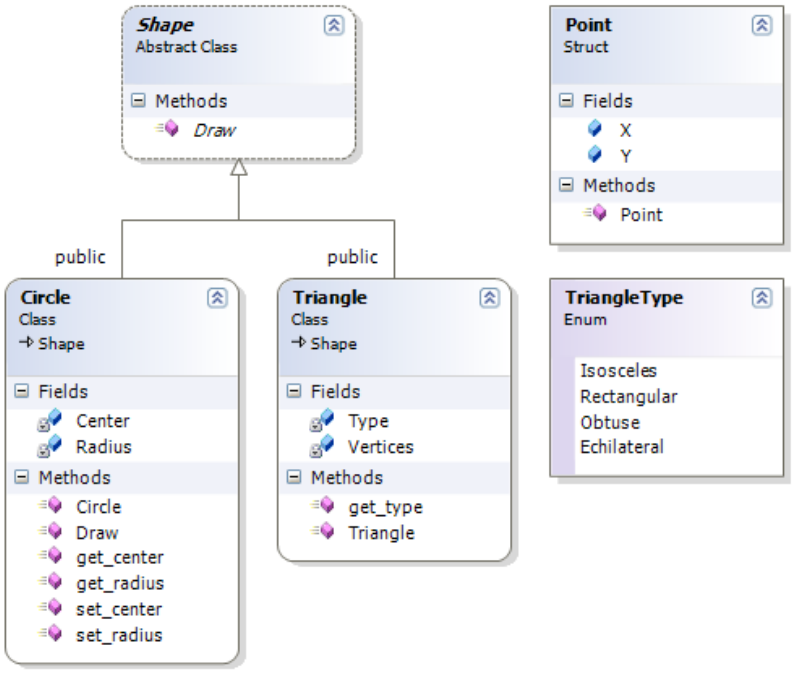


Figure 2.3: An example of a class diagram in Visual Studio (from http://mariusbancila.ro/blog/wp-content/uploads/2007/09/vs2008_classdesigner.png)

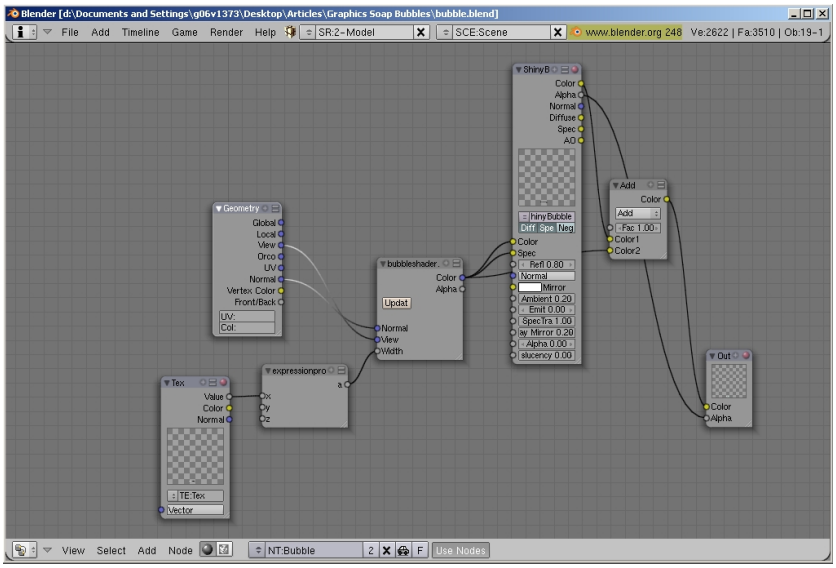


Figure 2.4: Visual editing of a Python shader in Blender (version 2.48).

2.1.5 Room for improvement

2.1.5.1 Some advantages of text

Text is an effective representation when implementing routines and the actions that take place in an algorithm.

- It can be quickly typed
- It can be compact and concise
- The meanings of many distinct words have already been memorised, whereas the same is not always true with colours or icons
- Blocks of text can be summarised as a single word or identifier (in the case of functions)
- Text is widely used and supported, and so a number of tools and facilities exist that make text simple to work with. Graphical decoration of text and a large variety in fonts make text even easier to read.

2.1.5.2 Some disadvantages of text

But text does have failings. There is no way to automatically adjust the level of detail (or zoom in and out) with text. Code folding achieves a fair approximation, and changing the font size is another (though not very useful) option, but there is no way to dynamically fold code, gradually reduce subsections to icons or ellipses, and so view more code at a higher level of abstraction. This is possible with a text editor, but not ideal, and could be better simulated by giving the underlying structure a graphical and two dimensional basis.

Text is serial and this greatly reduces complexity when deciding where to start and where to end reading. But this poses a problem when trying to simultaneously view non-adjacent portions code. It is sometimes possible to open multiple sub-windows and lay them out side by side. Often this requires using external text editors (if the IDE allows only one instance to run at a time, and especially if the code is in the same file which is then read-write locked by the first IDE). This means forfeiting the advantages of the IDE (syntax highlighting, auto-completion, class and method summaries) when working from any but the primary window. Also as each window is given focus, it covers over all other windows and complex juggling and scrolling about is required to manage the multiple windows and lay them out in their own space on the screen. This can be avoided by incorporating the use of multiple windows into the underlying structure of the IDE.

A number of techniques are used to make code easier to find, such as text search and summaries. But these mechanisms could be further improved by using spatial awareness and memory as an aid to finding locations of interest, and a greater use of visual cues such as colour or shape to tag currently relevant information.

Chapter 3

Related Work

3.1 Introduction

3.1.1 Visual programming

Visual programming was a major field of study between the mid-eighties and mid-nineties **(author?)** [10]. As computer hardware advanced, so did the ability to render and display graphical content. The proponents of visual programming hoped to harness this technology, and so advance the techniques used to program computers from simple text to present more natural methods. The aim of this endeavour was to find more effective representations to use in the translation between solutions and their programming language encodings **(author?)** [2].

The field of visual programming encompasses a number of different areas of research. Some researchers aim to produce graphical front-ends for existing textual languages. Others create entirely new languages based on diagrams and graphics. Some abandon textual encoding completely and spatially parse pictures and graphics as code.

This project takes the former approach, as designing, writing, establishing and maintaining a mainstream programming language is far out of the scope

of this project. Instead the approach has been to use existing and mature textual programming languages, and to build on these to investigate and demonstrate how alternative representations may be beneficial.

3.1.2 Dataflow

A large number of visual programming languages and interfaces use a representation known as dataflow. A dataflow diagram consists of a number of nodes linked into one another, where nodes receive input, do some processing and possibly produce output to send to other nodes. This is used to represent functions, where each node is a routine and may in turn be defined by a number of other routines. The input and output of these routines is combined to form a program. This has been used to model interaction with interfaces, where the program moves between different states based on what the user interacts with (such as Libero, available at http://legacy.imatix.com/html/libero/). It is also commonly used in graphical art software, where nodes perform different transformations to an image, such as adding colour, distorting the image, applying a mask or output the image to file (for example MapZone, available at http://www.mapzoneeditor.com/).

This project has not chosen this route for a number of reasons. First, this representation is fairly common, with a number of systems based on dataflow, and it is unlikely that it would have yielded much that is new. Secondly, at the level of routine implementation text works remarkably well. It is very compact, textual characters are easy to identify, it is possible to type very quickly, text is well supported by existing software and people have a lot of experience using text, it is simple (serial) and so easy to read. But to express a large amount of detail with graphical representations can result in a bulky, difficult to read diagrams, where each change may require rearranging the entire scene to make it readable. In more complex diagrams, it may difficult to find where to start and where to end.

3.1.3 Project focus

Instead this project focuses on areas where text does not do quite so well - notably that it is not as easily scalable and the serial format is in some cases too restrictive. As such, this project provides a high level front-end interface representing the data structures of existing textual languages, but leaves the low level implementation of algorithms as text. This excludes much of the research in visual programming, including graphical representation of algorithms, derivation and parsing of new graphical languages and the use of visual programming in education. Despite this, some of these papers have presented interesting questions, arguments or observations relevant to the current study, some of which are presented here. Other relevant papers have focused on issues such as how users perceive and interact with programming languages, considerations important when designing interfaces to programming languages and deciding on means of evaluating such interfaces

3.2 Contributions from existing systems

3.2.1 Introduction

Before designing a system it is important to look at what has gone before, to avoid pitfalls and wasteful reinvention, and to gain inspiration from the ideas of others. A number of visual programming systems have been developed and discussed in the literature. These have introduced a number of different ideas and been designed to fulfill a number of different aims. Some are front-ends for textual languages, some are languages designed to take advantage of graphical representations. Some are meant to make programming easier and more accessible to novices or non-programmers, some service specialised fields such as audio or electronics, some focus on data structures while others focus on algorithms. This section introduces some of the ideas and arguments that factored into the design of various systems, and some comments or criticisms

of various ideas and design decisions.

3.2.2 Write

3.2.2.1 The value of text and graphics

One of the data flow oriented visual programming systems that has been created is Pict. Pict was designed to run without any use of text or the keyboard, the paper's unsubstantiated claim having been that graphics are inherently better and easier to process than text **(author?)** [5]. A more realistic view is posed by **(author?)** [6], namely that text and diagrams each have different strengths and weaknesses.But a valid point raised by **(author?)** [5] is that graphical symbols may be used to compress a lengthy textual description into a compact graphic. But **(author?)** [11] has raised the equally important point that the interpretive value of images is not always beneficial, and can lead to ambiguity. With this in mind, it is possible to use graphics effectively to express large amounts of information in a limited space, but that the meaning of such graphics must be clearly defined to avoid ambiguous interpretation.

3.2.2.2 Importance of evaluation (author?) [3]

Tinkertoy is a graphical data flow oriented front-end for Lisp. An important observation made was that users will not work with anything more difficult to use than what they currently have. This highlights the importance of evaluation and comparison with existing means of programming, to determine whether improvements have been made. This is conducted in the evaluation section.

3.2.2.3 Role of an interface to code

A valuable contribution arising from the added interface between the user and code, is that systems such as Tinkertoy ensure that syntax mistakes can

never occur. Tinkertoy uses the powerful concept of context-sensitive menus attached to icons, meaning that the interface allows only legitimate actions to take place **(author?)** [3].

A criticism of both ThinkPad and Tinkertoy is raised by **(author?)** [12]. This paper claims that visual front-ends should completely hide the textual representations they abstract. But this seems unnecessary for two reasons. Firstly, users experienced with textual languages could use the textual representations as guides. Secondly, as each representation may be better or worse at expressing different concepts **(author?)** [6], the textual representations may be helpful where graphical representations are ambiguous, complicated or do not sufficiently emphasise the required information.

3.2.3 Navigate

An interesting idea raised by **(author?)** [2] is that graphical representations should act in a way with which users are familiar. For instance, double clicking on a class should open a window to view its contents, analogous to double clicking on an operating system folder. This metaphor could be extended in a number of ways. For example, file extension could be used to represent variable or method return type, and shortcuts used as pointers. Read and write access rights could represent the availability of get or set methods. Likewise, these virtual folders could be dragged, copied or renamed using the same shortcut keys or techniques used on operating system folders. But as this project focuses on higher level structures and abstractions, these method level ideas have not been implemented.

3.2.4 Understand

3.2.4.1 Interpretation

(author?) [11] has given a comprehensive overview of the issues relating to graphical programming and the setbacks that have prevented its dominance

over textual programming. He poses that text is more suited to programming, as programmers need a precise, almost mathematical, notation to unambiguously instruct the computer how to go about a task. This is opposed to graphics, whose ability to convey multiple and complex concepts using limited space and complexity is due to the interpretation afforded by pictures. A problem with graphics is then the issue of ambiguity, that different people may interpret the same picture in different ways. This is valid but ignores an important issue. Both text and diagrams may be susceptible to ambiguity and to interpretation according to the context in which they are used, for instance poetry, art or satire. To say that graphics are expressive only because of the number of ways of interpreting the same image, and so must be inherently ambiguous to be of any use, is misleading. Both text and graphics may be clear or ambiguous depending on the rules governing the language or diagram. The words used in programming languages are restricted and strictly defined for that language. In the same way unfettered graphics are open to interpretation. To use graphics in programming, the images would likewise need to be restricted and their usage and meaning strictly defined. But some problems or ideas are better expressed with the structural freedom of graphics, and others with the descriptive properties of text. Yet other problems or ideas require ambiguous interpretation (though not as likely for the cause of programming), but this is neither a property only of graphics nor the only property of graphics.

3.2.4.2 Visual Cues

An important part of programming emphasised by **(author?)** [11] is that of layout, termed secondary notation. The readability of code is greatly enhanced or crippled by the appropriate or inappropriate use of layout. Indentation, spaces and conventions all help convey the meaning of textual code and was found by **(author?)** [11] to be a major distinguishing factor between novice and experienced programmers. With graphical programming

layout becomes an even greater issue due to the departure from text to a greater reliance on visual cues. For a graphical programming language to ease interpretation, focus would need to given to encouraging or enforcing a logical layout of graphics.

3.2.4.3 Graphics and expertise

A claim made by **(author?)** [5] is that visual systems are suitable only while learning to program, and that users should make the transition to textual programming as they gain expertise. This is in contradiction to the conclusion drawn by **(author?)** [11] that the extra room for expression provided by visual languages makes their use more complex for novices, and that expertise is required to use this room for expression in a beneficial way. Despite **(author?)** [5]'s view, this project intends to address some of the issues of experienced programmers using ideas from visual programming, with the view that only a relatively short period of a programmer's career is spent learning to program.

3.2.4.4 Graphical debugging and execution (author?) [5]

An interesting idea demostrated in Pict is that of graphical debugging. This allows the user to watch their code execute, as the sections of code currently under executed are animated by the system. Added to this, Pict can run incomplete sections of code, to allow design and testing without writing a full program. But as this project focuses on higher level structure, investigating these ideas was out of the scope of the current project.

3.2.4.5 Issues with graphical representation

A point highlighted in **(author?)** [3] is that visual representations of languages may become very awkward when over complex. It is this issue that features partly in the decision to leave routine level implementation of the

current system as text.

Another issue is one highlighted by ThinkPad, a data structure oriented front-end developed for Prolog. In ThinkPad, users may choose between a circle or a rectangle to represent a data structure. This would allow programmers to use different styles for different programs, and the lack of consistency could make reading such representations difficult **(author?)** [6].

3.2.4.6 Icons

Prograph is a data flow oriented system. In a paper written on Prograph (**(author?)** [2]), it is claimed that one of the advantages of graphical representation is that, for example, a variable of a bicycle could be represented as a bicycle. But this recalls an argument by **(author?)** [11] about interpretation - all too often such representations are more easily understood by reading the textual captions. Other issues involved are the time, inclination and secondary tools required to draw the pictures for the multitude of, sometimes very temporary, variables appearing in a typical project. Finally, not all variables have an associated graphical representation. To avoid these issues, it may be better to use a limited set of commonly used graphics, the meaning of which users are required to learn.

3.3 Evaluation

3.3.1 Introduction (author?) [4, 1, 7]

Four methods of evaluation are available in HCI. These are user studies, heuristic evaluations, cognitive walkthroughs and predictive evaluation. Traditionally these techniques are used to evaluate and improve upon a single interface, but in this case their use will be expanded to compare interfaces, and so evaluate the successes and failings of the project interface over traditional IDEs.

User studies are considered one of the most important studies when evaluating a single interface, as it exposes problems that result in the field with real end users, rather than relying on predictions and hypotheses. However, a user study is less useful when comparing interfaces. It also has a number of drawbacks which proved insurmountable in this project and so was not used. These points are discussed below.

A heuristic evaluation is a discussion of guidelines, set out in the literature, which work as a checklist for evaluating the design of an interface. Such evaluations are conducted below on both tradition IDEs and on the project interface, to evaluate how each interface fares and highlight where, and where not, the project interface improves upon that which already exists.

A cognitive walkthrough is a test which relies on a number of scenarios, and evaluates how closely the solution to a real world problem maps to a description in some other format, such as a computer language. This evaluation helps to highlight distracting and unwanted steps that are required by the description format but are irrelevant to the problem being solved. As the interface is an extension of an existing language, and evaluation of the underlying language is not part of this project, this technique is less useful to evaluation of the project interface. Where it could feature is highlighting how distracting format-specific details are automated, and so removed or hidden from the user's perspective, and so mapping between the problem and the encoding is improved by the project interface. But due to time constraints, a cognitive walkthrough was not conducted.

Predictive evaluation is an objective form of evaluating the details of an interface. This again relies on a set of scenarios (which must themselves be guarded from subjectivity), decomposed into a sequence of basics steps formulated in the literature. Each step has an associated time value, such as the time taken to press a key or click a mouse. The total time taken to achieve the same goal on different interfaces is used to assess the efficiency of each interface for different tasks. For this project, these scenarios consist

of various programming tasks which the author believes to be common. The efficiency predicted for traditional IDEs is compared to that of the project interface, and is used to evaluate the project interface's success.

Select of these evaluations have been used to compare traditional IDEs and the project interface, and so evaluate where the project has made improvements over traditional IDEs, and where not. They have also been used to uncover other problems and potential improvements to both sets of interfaces. Rather than comparing a single IDE or the project interface as it stands, comparison has been made between features of the two that exist in at least one traditional IDE, or that have been considered for development in the project interface but might not yet have been implemented. If a discussed feature of the project interface has not yet been implemented, it has be noted as such.

3.3.2 User study (author?) [4]

In a user study, testers from the target user group are set tasks demonstrating different features of the interface. Their use of the interface is observed to uncover where difficulty or confusion arise, where mistakes are made and where improvements could be made. These results and feedback from the user are used to identify and improve on these flaws. For a comparative evaluation between two interfaces, such as the project interface and traditional IDEs, timings could be taken to provide a measure of comparison.

The advantage of a user study is that it is tested by real users - the issues uncovered are those that would be encountered under normal use, rather than predicted or hypothesised issues that might never arise in practice. It may also uncover issues that were missed during the other forms of evaluation. But there are a number of issues that made a user study infeasible for this project:

- Lack of testers: According to **(author?)** [4], a minimum of ten users are required in a user study to provide statistical significance, although

some researches claim only three are needed. Added to this, a number of users would be required for preliminary runs used to design the experiments. These results would be discarded and the testers could not be reused due to the resulting bias of having performed the tasks before. This number of testers were not available.

- Time: Developing, testing and collating suitable tests and tasks used in test scenarios would take a significant amount of time, which was not available. Participating in tests would also require a lot of time from the participants, which was also not available. This issue might be possible to circumvent if the participants were remunerated in some way, but this could lead to bias as the testers would be positively predisposed towards the test.

- Bias: User testing is potentially very subjective. Besides timings, results rely on questionnaires based on user opinion. This is useful in finding flaws and room for improvement, but less useful for comparative evaluation. The project interface would be novel to the use and may bias opinion either way. Order of testing and discrepancies and similarities between the test scenarios would be an issue. Users might be more capable when completing the second test due to their experiences in the first. Users might find the difficulty of different tasks to be unbalanced, and the affect of the interface would be skewed due to these results. There may also be bias due to the degree of familiarity or aversion to programming language or IDE used. Added to this, the user study would necessarily test novices of the system, whereas it the ease of use to experienced users that is more important (users are only novices for a short period of their use of the system). Correcting for bias is non-trivial, and would require assessing the predisposition of the user and using this information to temper the results. The order of tasks, and assignment of tasks to a traditional IDE or the project

interface could be inverted between groups of testers, but this would again require a large number of testers and a lot of time to collate and analyse. As an example of these issues,**(author?)** [5] evaluated Pict using an opinion survey. A number of issues about these tests were raised, such as the role novelty of the system may have had, and the presence of Pict's author during testing.

- Redundancy: The same comparative results (timings) can be predicted objectively using a predictive evaluation.

3.3.3 Heuristic evaluation

A heuristic evaluation is essentially a discussion of a set of guidelines, set out in the literature. These guidelines serve as a checklist of areas where users may have difficulty, of pitfalls that should be avoided and of additions that serve to improve interfaces. These are used to identify problems with the interface, and is especially useful in identifying discreet issues which a user may find a hindrance but not consciously notice, or may be unable to describe. They also set out the path towards a good interface, uncovering areas which might not be an active problem, but could yet be improved upon.

These guidelines are to compare two interfaces - the traditional IDE and the interface developed in the course of this project. There are a number of sources for these guidelines. Many of these guidelines are too general, too specific, or are otherwise not applicable to the current evaluation. A set of heuristics by Gerhardt-Powals, as given in **(author?)** [4] have been used in this evaluation, as there are a manageable number of them and they discuss issues related to the aims of the project. A second set of heuristics by **(author?)** [6] has been chosen for this project, as it is aimed specifically at visual programming interfaces.

3.3.4 Predictive evaluation

3.3.4.1 Introduction (author?) [8, 7]

Predictive evaluation allows the analyser to predict the performance and efficiency of interfaces when used to perform different tasks. The advantage of predictive evaluation models is that they can be conducted before the interface is complete. They do not require training in psychology in order to conduct. Importantly the results they yield are objective, and the steps taken are open to criticism and correction. The restriction of these models is that they predict only error-free expert use. But as this system assumes expert use, this is an advantage.

The most mature of these evaluation models is the GOMS family, which is an acronym for Goals, Operators, Methods and Selection rules. The GOMS model allows the user to break down goals and tasks into a set of subtasks. These subtasks consist of pre-defined operators. Existing research on these operators allows the analyser to make predictions about the tasks comprising them.

A number of different forms of GOMS evaluations are available. Some are more complex than others. With this complexity come extra predictive abilities, such as predictions of the learning curve and of errors that are likely to occur. The simplest GOMS model is Keystroke-Level Model GOMS (KLM-GOMS). This satisfies the most important goal of the evaluation, namely predicting the traditional and project interfaces' efficiencies. This is the model used in the study, as the benefits of using one of the more complex models are low and the increased complexity gives rise to an increased probability of errors.

The process of conducting a KLM-GOMS evaluation consists of identifying the main tasks in the system. These are then repeatedly broken down into subtasks, which in turn consist of operators or more subtasks. Finally, the timings for each operator are used to calculate the time taken to execute

each task.

3.3.4.2 Operators

KLM-GOMS consists of a number of operators and their associated timings. The recommended times for each operator were used in the study. These are repeated below, as taken from **(author?)** [9], but with comments derived from **(author?)** [1]. The times given in **(author?)** [9] differ slightly from those in **(author?)** [1], but **(author?)** [9] is far more recent and, although the report was not published, the author is a major contributor to the field of GOMS modelling. The shorthand symbol used to represent each operator is given in brackets:

- Pressing or depressing the mouse button (B) - the recommended time is 0.1 seconds.
  - It then follows that a click is BB, and double clicking is BBBB, also written as B(4).
  - Separate times are not given for rolling the mouse wheel. This action is assumed also to amount to a B.
  - Where the number of presses or scrolls are case dependant, assumed values are given to simplify the analysis.
- Homing (moving hands to) a device, in this case the mouse or keyboard (H) - the recommend time is 0.4 seconds.
  - As a simplification, it was assumed that either the mouse or the keyboard were used. But in some cases a user could have one hand on the mouse and the other on the keyboard.
- Pressing a key (K) - the recommended time is 0.28 seconds.

  - K is not divided into a press and depress, as is the case with a mouse button press.
  - For variable string lengths, assumed values are given to simplify the analysis.

- Mentally preparing for an action (M) - the recommended time is 1.2 seconds.
  - This time is not accurate if, for instance, the user needs to think up an name or perform some other creative activity. Such special cases cannot be accurately predicted, and as a simplification such cases were ignored.

- Point or move the mouse cursor onto a target (P) - the recommended time is 1.1 seconds.
  - The times taken when pointing or steering can be calculated using the equations known as Fitt's Law and the Steering Law respectively. These calculations take into account the size and distance of the targets. This information is partly available, but differs from operating system to operating system, depends on the widget set settings, and depend on the screen resolution. As a simplification, both steering and pointing were assumed to have the default value.

- System response time (R) and wait time (W) - there is no recommended time.
  - These times must be determined on a case to case basis. In the case of the response time, it is assumed that the platforms on which the systems operates are fast enough to run without any

delays. But wait time is significant when scrolling over long distances, where the user must wait to arrive at the destination. As a simplification, it is assumed that such navigation techniques are used only to reach adjacent portions of the workspace, and so require no wait time. Fortunately there are alternative navigation techniques for both the traditional and project interfaces which do not require waiting and have efficiency times close enough to make them viable substitutes.

3.3.4.3 Rules

A number of rules are given which decide when and where operators are required. These rules, taken from **(author?)** [1], are explained in the context of this evaluation below:

- Rule 0: Insert M's in front of P's, B's and K's. For example pointing the mouse and depressing the mouse button would at first glance be MPMB.
- Rule 1: If one operator fully anticipates the next (there is no need to think about what to do next), then remove the M between them. For example the action above would then be MPB.
- Rule 2: If the a string of key presses is part of the same cognitive unit (i.e. there is no need to think about which letter comes next, as in the case of a command name), then delete all the M's except for the initial one. For example "open" would go from MKMKMKMK to MKKKK, also written as MK(4).
- Rule 3: If K is a redundant terminator, for example a terminating return key press immediately following another terminating return key press, then remove the M in front of it.

- Rule 4: If K terminated a constant string, for example a return key press after a reserved word, then remove the M in front of it. But if it comes after a variable string, for example a return key press after an identifier, then do not remove the M.

3.4 Summary

Numerous examples of visual programming software for audio programming, electronic circuit design, education and graphical design applications appear frequently in a web search on the topic. But examples of visual programming concepts in mainstream use for general programming are more difficult to uncover. A number of purely visual languages and interfaces have been developed in the past, but now largely exist only in the literature. Prograph is one of the rare examples that has survived, now marketed commercially as Marten (http://www.andescotia.com/) and available for MacOS.

Despite the lack of mainstream success of these systems, their development nonetheless generated many interesting ideas from which this project has drawn inspiration. But before designing such a system, it is important to have an appreciation of the issues involved, and a set of goals towards which development is aimed. Insightful papers such as **(author?)** [6] and **(author?)** [11] provide many arguments and interesting points for consideration. Constantly aiming toward these goals, and carefully avoiding the mistakes and pitfalls of the past, is important when the aim is to yield more successful techniques of programming than those currently available.

A number of methods of evaluation are available, including user studies, heuristic evaluations, cognitive walkthroughs and predictive evaluations. Of these, predictive and heuristic evaluations have been used to evaluate the project. The predictive evaluation provides numerical results to evaluate the interfaces' efficiencies, while the heuristics are used to evaluate and discuss the more abstract components of the interface.

Chapter 4

Design

4.1 Introduction

4.1.1 Components

This project provides an interface to represent code in a format which is easier to write, navigate and understand than current textual representations. This is achieved by using a number of components in conjunction with one another. These components may work individually or in combination, each addressing part of the above aims. An outline of these components and how they satisfy the project aims is given below.

4.1.2 Design outline

4.1.2.1 Writing

- Creating data structures
  - Class trees
- Editing data structures
  - Text windows

4.1.2.2 Navigating

- Defining the viewspace
  - Scrollwindow
  - Workspace
- Moving the viewspace
  - Zoom
  - Minimap
  - Scrolling

4.1.2.3 Understanding

- Explaining function
  - Code tour
- Showing associations
  - Class trees
- Summarising and abstracting information
  - Minimap
  - Zoom
- Hiding information
  - Scrollwindow
  - Scrolling
  - Text windows

4.2 Components

4.2.1 Introduction

This project consists of a number of components, each of which contributes to achieving the project aims. This section introduces the components, what they are, what they do and how they contribute to the system. Some definitions are used frequently in the text below:

viewspace the portion of the workspace currently displayed on the screen.

workspace the entire project represented in two dimensions.

4.2.2 Workspace

4.2.2.1 Introduction

The workspace consists of the class trees of the entire project, arranged on a two dimensional plane. The user then decides which portion of the workspace to view and at what level of detail.

4.2.2.2 Contributions to navigation

The workspace merges the information that would usually be presented in multiple files, and presents it on a single plane. This helps avoid user disorientation from jumping to and from different files. As it is in two dimensions, the user can use his spatial awareness to arrange and find parts of the project, for example arrange classes according to their function or retrieving a class last seen in the top right corner.

4.2.3 Scrolling

4.2.3.1 Introduction

Scrolling is the ability to move the viewspace, namely the portion of the workspace currently displayed on the screen.

4.2.3.2 Contributions to navigation

Scrolling allows the user to maintain his level of detail, but to view different parts of the project. For example, the user may wish to view the details of the classes on which he is working, but the classes do not fit together in the viewspace. He can then scroll from one class to the other to bring each into the viewspace as it is needed.

4.2.3.3 Contributions to understanding

Scrolling allows the move into the viewspace the portions of the project currently relevant, and so reduce the amount of detail on the screen.

4.2.4 Class trees

4.2.4.1 Introduction

A class tree is a graphical representations of a hierarchy of classes (Figure 4.1 on page 33). This consists of a number of nodes (graphical representations of classes). After these have been zoomed in past a certain threshold, they display their internal details (4.2).

4.2.4.2 Contributions to writing

The trees automate a large amount of the workload that would normally be handled manually by the programmer. This avoids the user having to manage files, input keywords or enter the name of the parent class.

4.2.4.3 Contributions to understanding

The trees show associations between classes. This is achieved by highlighting any classes which are contained as fields in the currently selected class. The trees also manage comments by displaying the comments associated with the selected class in a separate text window (4.3).

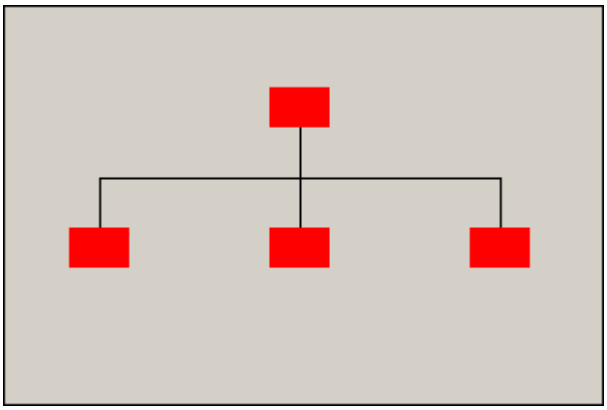


Figure 4.1: A class tree.

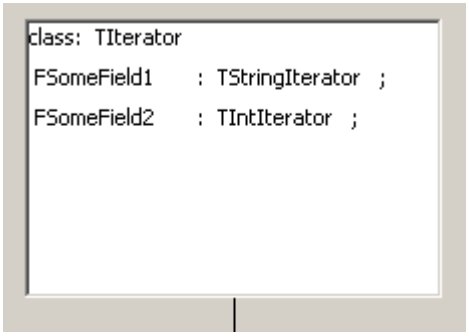


Figure 4.2: One of the class nodes zoomed in and showing the internal details.

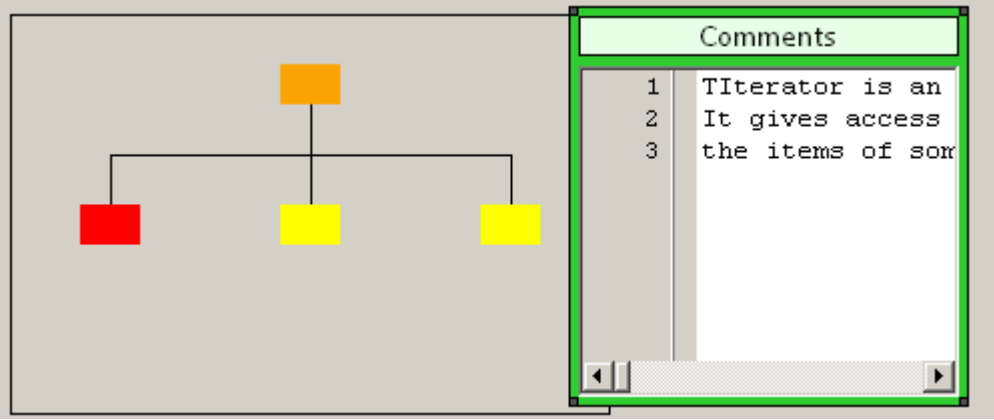


Figure 4.3: The highlighted class node (top node, orange) contains fields which are instance of the yellow highlighted nodes (two bottom right nodes). By selecting the class node, any comments associated with the node are also shown.

4.2.5 Zoom

4.2.5.1 Introduction

Zooming is the animated scaling of objects, to allow the user to see an overview of the entire project or a detailed view of only a portion. For example, 4.2 is more zoomed in than 4.1.

4.2.5.2 Contributions to navigation

Zooming aids navigation by allowing the user to zoom out until their destination fits within the viewspace. From there they can zoom in on their destination. By allowing the user to fit more of the project inside the viewspace, the user can determine the spatial relationships between different structures.

4.2.5.3 Contributions to understanding

Zooming allows the user to view information at the level of detail currently desired. They may either view a summary of a large portion of the project,

or the details of a small portion of the project.

4.2.6 Text windows

4.2.6.1 Introduction

Text windows are moveable text boxes used to display code, comments or descriptions. An example is the comment box in 4.3.

4.2.6.2 Contributions to writing and navigation

Text windows are spatially disjoint and can be freely moved around, as opposed to text which is serial an cannot be moved out of sequence. This allows the user to drag different text boxes into the proximity of one another, for example if the user wishes to work one three different methods. This avoids the necessity of jump to and from different files.

4.2.6.3 Contributions to navigation

As the details of code, such as comments or method implementation, are contained in text windows, the user can choose to show only the text currently relevant, and hide all other text windows. As the text boxes may be moved around, the user can spatially arrange the text boxes in the viewspace according to his own method of grouping, such as according to their contents or current priority.

4.2.7 Minimap

4.2.7.1 Introduction

The minimap (4.4) gives a second and higher level view of project. This allows the user to get an abstract overview of the project from the minimap, while at the same time working on a more detailed portion of the workspace in the scrollwindow.

4.2.7.2 Contributions to navigation

The summarised overview of the minimap allows the user to know where they are in relation to the rest of the project, providing orientation. It also provides coarse navigation from anywhere to anywhere else.

4.2.7.3 Contributions to understanding

The minimap gives an overview of the spatial relationships between objects on the workspace. This may aid understanding if the user has attached any meaning to these relationships, such as grouping class trees by function.

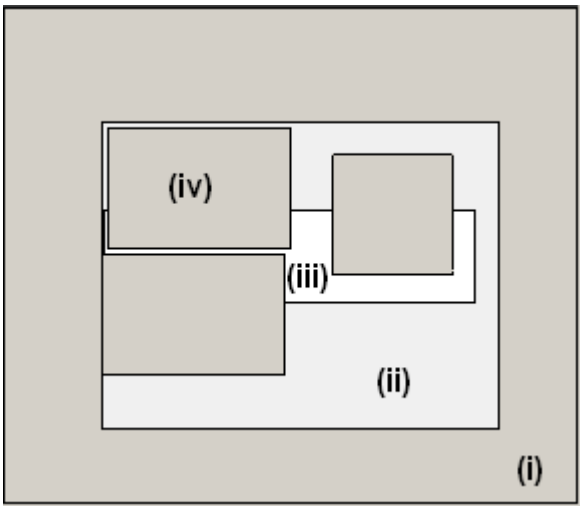


Figure 4.4: The minimap. (i) Outside of the project. (ii) The workspace. (iii) The viewspace. (iv) An object on the workspace (in this case a class tree).

4.2.8 Code tour

4.2.8.1 Introduction

A number of tools exist to help document code. Often, these tools parse code and extract text within special documentation code blocks. These com-

ments are then formatted and presented in html or pdf documents. In some IDEs, these comments are displayed in a hint when the mouse hovers over a relevant keyword - and instance of the class or the use of a function. Sometimes accompanying diagrams are provided to illustrate the structure and relationships between parts of the project.

The code tour attempts to improve on these mechanisms by providing a more interactive and integrated means of documenting code. The code tour visits a list of destinations or waypoints, such as the position of a class or class tree, which can be visited in any order. At each waypoint, a predetermined action occurs, such as displaying a textual explanation, a diagram or an animation. This is used to explain to the viewer the function of the code.

4.2.8.2 Contributions to understanding

The code tour allows the documentor to associate explanations with the code itself, rather than in separate documents referring to the code. It also allows more interaction between the user, documentation and code - such as viewing only an introduction to each section of the tour, or viewing on sections of interest in detail, or using the code tour to write a program step by step as part of a tutorial.

4.2.9 Scrollwindow

4.2.9.1 Introduction

The scrollwindow (4.5) contains the viewspace, and allows the user to scroll relative to their current position, such as up, down, left or right.

4.2.9.2 Contributions to navigation

The scrollwindow allows for fine grain navigation, relative to the current position. This is opposed to the minimap, which navigates at a higher level

of abstraction and can jump from and to anywhere. This is useful for visiting portions of the project adjacent to the current position.

4.2.9.3 Contributions to understanding

The scrollwindow display the details, as opposed to the minimap which displays only an overview.

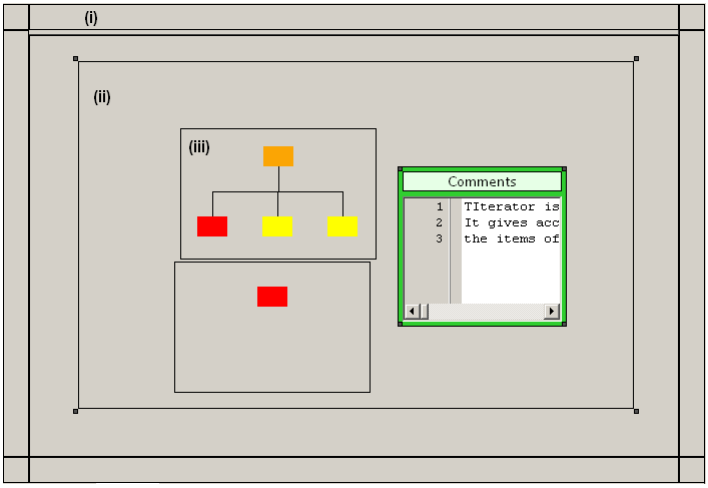


Figure 4.5: The scrollwindow. (i) When the mouse moves over the border, the scrollwindow scrolls in the associated direction. (ii) The workspace. (iii) A class tree.

4.3 Summary

The above components form the redesigned interface used to address the shortfalls of the traditional IDE. How these components function is given in more detail in the next section, on implementation.

Chapter 5

Implementation

5.1 Introduction

5.2 Tools

A number of mature tools, IDEs and libraries are available to aid GUI development. These include the C++ toolkit QT, the Java IDEs Netbeans and Eclipse, the C++/C#/Visual Basic IDE Visual Studio and the Pascal IDEs Delphi and Lazarus. Although the project would have been possible with any of the above, the Lazarus IDE was chosen due to the tight integration and support for GUI development provided by the IDE.

5.3 Project interface

The aim when implementing the interface was to provide a number pluggable utility classes, and combining these to add services, such as zooming, moving or scrolling, to the system components.

5.3.1 Utility classes

These classes perform a large amount of the work of the other classes. These consist of:

Notifier This class provides communication between classes, by notifying other classes of some change or event. For example, when the scale changes, the bounds of all affected controls must be resized. When the scrollwindow scrolls, other components such as the minimap need to be updated.

Scalers These classes provides scaling for other classes. For instance a scaler may be used to maintain the bounds of a control according to some scale value, or a list of these scalers may be used to simulate scaling of all the controls in a window.

Mover This class manages moving controls around the screen according to mouse movements. Classes using the mover tell it when to activate or deactivate, and the current position of the mouse. The mover then calculates the new position of the control and moves it there.

Resizer This class attaches and manages resize handles to a control. When these handles are moved, the resizer determines the new bounds of the control and resizes it.

ScrollPlugin This class uses a given position and destination to determine which direction to move a control, then moves the control in the calculated direction whenever a Timer fires Alternately it can use a list of waypoints to determine the destination, and fire an event when the destination is reached.

GlobalData These objects contain data shared amongst other controls, and use Notifiers to communicate with other objects when this data is changed - for example the default colour or how a control should react when

interacted with by the user. This allows the user to change the display and behaviour of various classes from a central point.

BoundsManager This object clamps values between fixed or calculated bounds. This is used by a control to determine where it is legal or illegal to move.

5.3.2 Managing state

These classes managing the transition of controls from state to state according to the way the user interacts with them, and changes the display according to the state.

ControlState This class determines holds the current state of a control. When the user interacts with the control, this is relayed to the ControlState which decides whether the interaction is legal and how it affects the current state.

Painters These classes draw on other controls. A subset of these classes use the programs state, as contained in the ControlState object, to determine what to draw on the control - for example what colour or which image to display.

BlankControls These controls do not perform any of their own state or display management, but defer these to ControlState and Painter objects.

5.3.3 Program structure

These classes are used to represent the project code.

ClassTreeGUI This control uses a basic tree structure to determine what classes are in the tree and where. The TreePlotter is used to arrange the nodes of this tree spatially.

TreeNodeGUI This control consists of a number of two parts. The borders which determine whether to add a child, left sibling or right sibling. The central ClassGUI is used to graphical represent the class contained at a node.

ClassGUI This control has a references to the details of the code it represents. When the ClassGUI's scale is shifted past a certain threshold, it shows or hides a graphical representation of the code.

TreePlotter This class takes a tree and a number of parameters, such as the output tree's width, and determines where each node will belongs spatially.

5.3.4 System components

These are the core components of the system not discussed above.

TextWindow This control has a text box to display a string, and uses the utilities classes to resize and move.

Minimap This class uses the utility classes above to manage scaling and scrolling. It scans a given panel and determines where to place placeholder controls to represent the panels contents. For example, the minimap scans the workspace to determine the position of the ClassTreeGUIs.

ScrollWindow This class the utility classes to manage scrolling and scaling of a given control, which acts as the workspace.

CodeTour This is achieved through the ScrollWindow and the ScrollPlugin.

WorkSpace This may be any control the user decides, preferably one that can contain other controls.

LayoutManager This class calculates where new controls should be placed on the workspace, relative to some calling control. At present this just places the new control besides the calling control.

SystemInterface This class manages the rest of the system, such as choosing a TextWindow display comments or a LayoutManager to place a control.

Chapter 6

Evaluation

6.1 Heuristic evaluation

6.1.1 Introduction

As mentioned above, heuristic evaluations are used to discuss the components of the system, and are the only evaluation used for the more abstract aim of improving on understanding. Given below are guidelines drawn from two sources. The first, from **(author?)** [4], are general HCI guidelines that apply to this project. The second, from **(author?)** [6], are guidelines aimed specifically at visual programming systems.

6.1.2 Write

6.1.2.1 Automate unwanted workload (author?) [4]

The project interface manages and automates code generation and file management, whereas on a traditional IDE these are mostly manual, although with the help of autocomplete, wizards and templates.

6.1.2.2 Style of development(author?) [6]

Not all programmers are the same. Not all projects are the some. Some users or projects are more disposed to top-down editing, others to bottom-up editing, and there will more than likely be a mixture of both.

When designing the interface, users should not be constrained to a pre-defined style. Neither should they be hindered from changing styles. Ideally, both top-down and bottom-up editing should be possible from the same view, to avoid the user having to re-orient himself at every switch. As shown in the predictive evaluation section, this interface does offer some bias by better supporting some styles over others. But the user may switch between the interface and traditional textual code to avoid these restrictions.

6.1.2.3 Code reuse(author?) [6]

Programmers often reuse old code, possibly modifying it in some way. Code is copied and modified by users attempting to learn from an example, or using it as base for further development.

The interface should allow users to import, export, copy, integrate and remove portions of code with minimal hassle. These operations could be achieved using drag and drop techniques, but are out of the scope of the current project. Also, allowing the user to interact with the code at different levels, such as entire classes or methods, should improve over the character level text manipulation provided by most text editors.

6.1.2.4 Workspace(author?) [6]

Programmers generally write in small spurts, iteratively writing a bit here and something else there, then combining it all together again. It may be a hindrance to users if they have to flip between multiple pages during this process. Rather the window or workspace needs to be large enough to allow the user to jump between different parts of the project. Alternatively, they

should be able to bring the different pieces together so that they fit into the limited workspace. A similar situation arises where users search or browse information, such as to check dependencies make comparisons.

Again, a workspace larger than the screen addresses partially addresses this issue. But it may also be beneficial to create temporary mini-workspaces, where users can add or remove different views and components, then either delete or store this assembly for later reference. But these ideas have not been implemented.

6.1.2.5 Debugging program fragments(author?) [6]

To avoid bugs, programs are often written progressively by writing a small fragment, then testing it. Likewise, when debugging code is often pruned into small segments by commenting out surrounding code, to isolate the faulty piece of code.

As debugging is a major part of programming, it is important for the interface to facilitate this process. This could be aided by making debugging simpler, such as commenting out entire structures at a click or automatically commenting out all dependant code as well. Systems such as Pict **(author?)** [5] allowed the programmer to interactively and visually debug a program, and to run incomplete sections of code which halt when they run out of code to execute. Similar facilities may be replicated by integrating the environment with a debugger, but may not be as effective if routine level implementation remains textual rather than graphical. But method-level implementation is out of the scope of this project.

6.1.3 Navigate

6.1.3.1 Group data in consistently meaningful ways to decrease search time (author?) [4]

As the various components of the system are no longer serial, as in the case of traditional IDEs, the user can arrange objects according to his own categories.

6.1.3.2 Spatial reasoning (author?) [6]

When searching for information, users may use spatial orientation to find it, for example remembering that some item was last in the bottom right corner of the screen.

This concept is central to the design of the project. By presenting a single workspace, necessarily larger than the screen, the spatial reasoning of users is given more support than traditionally provided. The predicted time saved by merging multiple files is given in the predictive evaluation section.

6.1.3.3 Following paths through code(author?) [6]

Different environments provide different means to browse through code. But following a path through code, and importantly navigating back along this path, is often complicated. For instance, some editors allow the user to follow hyperlinks. But returning to the hyperlink may be difficult, and so IDEs often provide bookmarks. Bookmarks may also be used in the above example of visiting separate parts of a project.

Using a single large workspace avoids some of the disorientation caused by changing files, and a minimap is provided to allow global scale navigation and so better allow the user to orient themselves. In place of bookmarks, beacons or landmarks could be placed to remind users of where they are and where they wish to be.

6.1.4 Understand

6.1.4.1 Reduce cognitive load by bringing together lower-level data into a higher-level summation (author?) [4]

Zooming and the dynamic level of detail allow classes to be reduced to graphics, and a project consisting of many files to be summarised as one workspace. Traditional IDEs do offer some support through code folding and class summaries.

6.1.4.2 Present new information with meaningful aids to interpretation (author?) [4]

The code tour uses diagrams and animations to help the user understand the code.

6.1.4.3 Make appropriate use of colour and graphics (author?) [4]

The project interface uses graphics and colour to represent code, although this is partially true in traditional IDEs through graphical decoration of text.

6.1.4.4 Include in the displays only that information needed by the user at a given time (author?) [4]

The user can zoom, scroll and hide objects to ensure that only relevant parts of code are present. Traditional IDEs provide code folding for a similar purpose, but the ability to hide distracting details is limited.

6.2 Predictive evaluation

6.2.1 Introduction

As mentioned above, KLM-GOMS has been used to determine the efficiency of the project interface compared to that of traditional IDE interfaces. This

has been achieved by decomposing programming into a series of tasks, and calculated the predicted efficiency of performing the tasks on the project interface and a traditional IDE interface. The full workings are available on the accompanying CD-Rom.

6.2.2 Example

As an example of how this was achieved, given below is a worked example of how a class is created using the project interface (6.1).

2.5		**Navigate to class tree (Minimap click-scroll)**
	H	(Hand on mouse)
1.2	M	Think about where to click
1.1	P	Point to minimap destination
0.2	BB	Click
		(Hand on mouse)
5		**Create Class**
		(Hand on mouse)
2.5	Other	Navigate to class tree
		(Hand on mouse)
		(Hand on mouse)
2.5	Other	DoCreateClass
		(Hand on mouse)
2.5		**DoCreateClass**
		(Hand on mouse)
1.2	M	Locate class entry point
1.1	P	Point to class entry point
0.2	BB	Click
		(Hand on mouse)

Figure 6.1: The task (create a class) is broken into subtasks, namely navigating to the class tree and selecting where to add the new class. These are then broken down into KLM-GOMS operators (M, P, B, H) which are used to determine the time taken to achieve the task.

6.2.3 Navigation (Figure 6.2 on page 51)

6.2.3.1 Scroll components

- Text Editor
  - Scrollbar scroll
  - Mousewheel scroll
- Scrollwindow
  - Drag-scroll
  - Border scroll
  - Arrow scroll
  - Zoom scroll
- Minimap
  - Drag-scroll
  - Click-scroll

6.2.3.2 Other

- Text Editor
  - Open file tab

6.2.4 Writing (Figure 6.3 on page 52 and Figure 6.4 on page 53)

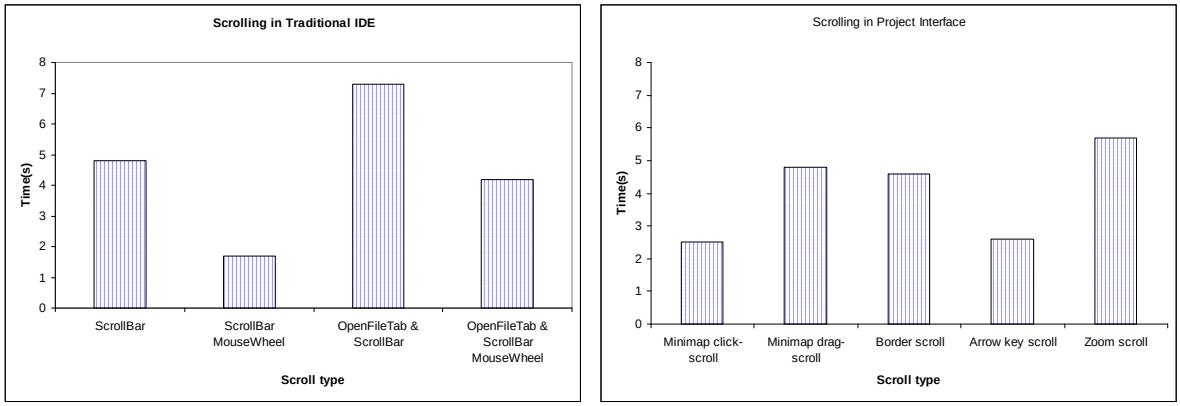
- Text Editor
  - Arranging text

  - Typing text

- Class tree
  - Adding class nodes
  - Drag and drop fields
  - Editing class fields

6.2.5 Results

6.2.6 Navigation



(a) (b)

Figure 6.2: Navigation using (6.2a) a traditional IDE with and without changing the current page in the editor, (6.2b) the project interface.

6.2.7 Writing

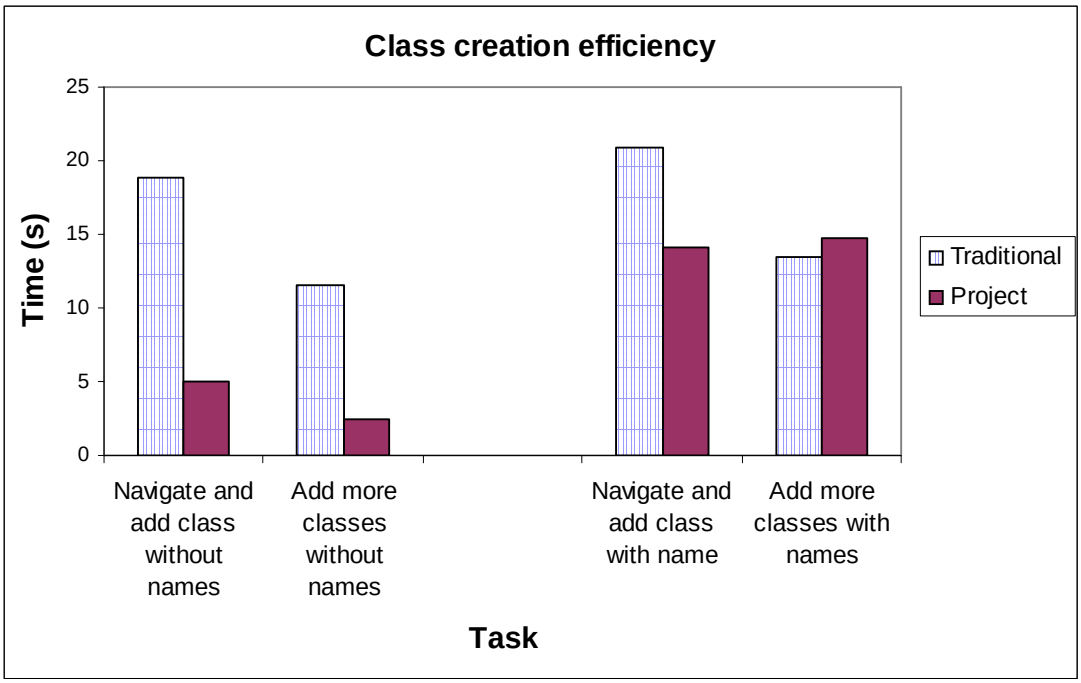


Figure 6.3: Relative efficiency when creating classes. On the left is creation of an empty class with no name, on the right is creation of a class and editing it to give it a name.

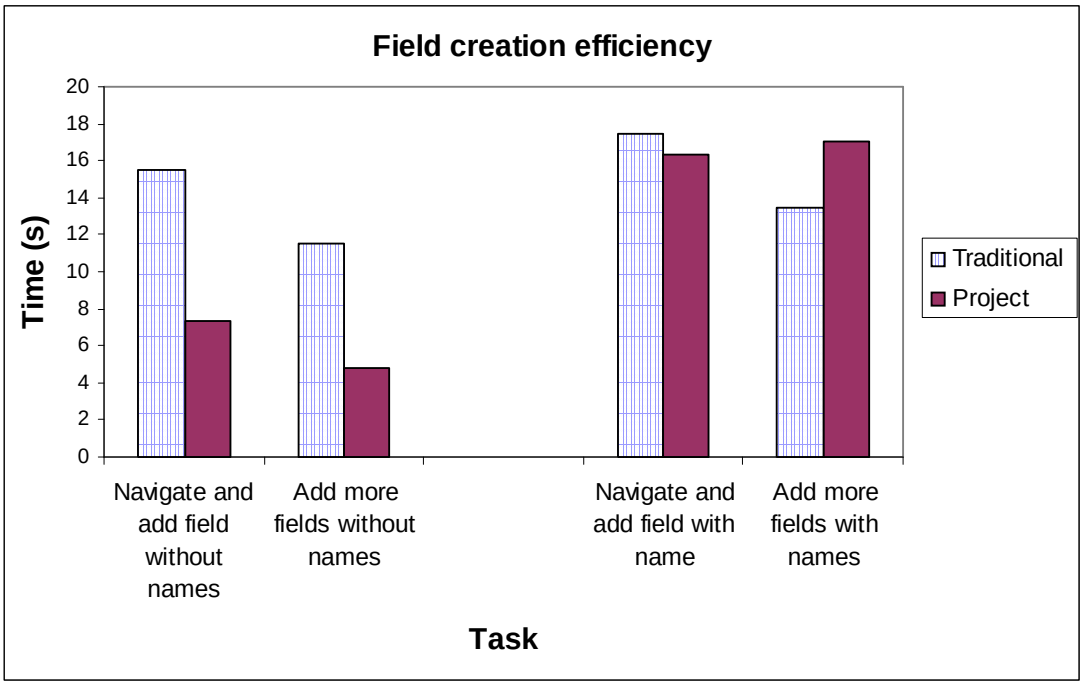


Figure 6.4: Relative efficiency when creating fields. On the left is creation of a field with no name, on the right is creation of a field and editing it to give it a name.

Chapter 7

Discussion

From the predictive results it is evident that it is possible to improve upon the efficiency of the IDE design. But the project interface provides most improvement when a specific style of coding is followed - namely creating a batch of fields or classes, then zooming in and editing all of them. When the contrary coding style of creating a class or field, zooming in, editing the class or field, zooming out and repeating the process was followed, the change in efficiency was minimal improvement or even a decrease in efficiency.

In this case zooming is an issue, as the user must zoom in and out to manipulate classes and to view their details. A possible solution to this is to display a separate window with the currently selected class showing it's details. If the user wishes to view the details of multiple classes simultaneously, he may then arrange them at adjacent positions and zoom in.

Unfortunately these differences in efficiency force the user to adopt a specific style, which goes against the relevant guideline given in the heuristic section. This is an area of concern with the project interface.

Chapter 8

Conclusions

IDEs began as text editors, and have remained this way. This project looks at improving this design. This redesign focused on representing the structures within code in new ways, on the premise that the majority of widely used programming languages are object oriented.

To achieve this, graphical representations of code structures have been created, and may be manipulated by moving, zooming and scrolling. These are placed on a single plane, to simplify navigation. A scrollwindow and minimap are provided to give views onto this plane at different levels of detail. A code tour has been provided to better integrate documentation with the code under explanation.

The effectiveness with which these components improve the efficiency of writing, navigating and understanding code was assessed using heuristic and predictive evaluations. These show that the interface is beneficial due to workload automation, the use of spatial reasoning, the simplifications due to a single workspace, the use of data hiding and abstraction, and the facility to include aids to interpretation alongside code.

But they also show that, provided the user need not first select a different file, scrolling through text using the mouse wheel is faster than any scrolling technique offered by the project interface. They also show that the project

interface is far more efficient when creating fields or classes, but only if these are created in batches and later edited. This strongly encourages the user to adopt a specific style, which goes against the heuristic guidelines.

From this, we have shown that

- IDEs have remained unchanged as text editors.
- Past attempts to redesign the IDE have focused on method level implementation, and have not been successful.
- This project has focused on representing code according to its structure.
- We have found our representation to be more efficient when creating and editing code structures in batches.
- But this efficiency is largely lost if the user wishes to create and edit structures one at a time.

Bibliography

[1] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, USA, 1983.

[2] P.T. Cox, E.R. Giles, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *1989 Proceedings of the IEEE Workshop on Visual Languages*, pages 150–156, Washington, DC, USA, 1989. IEEE Computer Society Press.

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

[4] Wilbert O. Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

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

[6] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[7] Bonnie E. John and David E. Kieras. The goms family of user interface analysis techniques: comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3(4):320–351, 1996.

[8] Bonnie E. John and David E. Kieras. Using goms for user interface design and evaluation: which technique? *ACM Trans. Comput.-Hum. Interact.*, 3(4):287–319, 1996.

[9] D. Kieras. Using the keystroke-level model to estimate execution times. pages 1–11, Ann Arbor: Department of Psychology, University of Michigan, USA, 2001. Unpublished report.

[10] B.A. Myers and A.J. Ko. The past, present and future of programming in hci. *Human-Computer Interaction Consortium (HCIC'09)*, 2009.

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

[12] G. Rogers. Visual programming with objects and relations. In *1988 IEEE Workshop on Visual Languages*, pages 29–36, Washington, DC, USA, 1988. IEEE Computer Society Press.

Appendix A

Accompanying CD-Rom

Contents:

- Demonstration of the project interface
- Source code of the project interface
- Working for the predictive evaluation
- Copy of this document