Matthew van Cittert 21 May 2009

Project Specification

1. Motivation

What is currently available

Code is written as text files. IDE's complement these text files, by adding facilities to read, edit and navigate code.

Variations of what popular IDE's and tools will do (Figure 1)

- Highlight parts of syntax to make it easier to read.
- Allow the user to follow hyperlinks to jump between definition and declaration.
- Automatically generate code outlines from templates.
- Find references to an identifier.
- Show base and derived classes and enumerate methods.
- Generate documentation from code and documentation comments, or from external files.
- Integrate test suites.
- Provide different views such as files or classes or methods.
- Summarise the classes present in a project.
- Highlight syntax errors.
- Highlight unused or uninitialised variables.
- Support code refactoring.
- Provide search and replace facilities.
- Allow restrictions of certain 'bad practice' code statements, such as goto's.
- Auto-complete code.
- Merge information from multiple files, simulating one large file.

What popular IDE's and tools will not do

- Allow more than a single column of text.
- Abstract different languages to a common representation.

- Service database style queries.
- Display or allow access to the intermediate code generated from macros or annotations.
- Give special facilities for patterns or API's.

Room for improvement

- Interactive testing.
- Shift of emphasis.
- Limit displayed information to currently relevant details.
- Use of space.

2. Proposed improvements

Allow more than a single column of text.

Multiple paths, in the case of conditional statements, and parallel execution, in the case of threads, may benefit from side-by-side representation. This is seen in Nassi-Schneiderman diagrams of if statements, and in diagrammatic explanations of parallel environments, such as GPU's. Preferably, all columns should be displayed simultaneously. In order to fit within the screen width, they would need to display only a summary or hint to their contents. Once the user has found the appropriate column, this could be expanded to view its contents.

This could be achieved in textual editors using columns or tables. But to express effectively it may require considerable graphical reinforcement, which the simple text editors found in IDE's do not support.

This approach partitions the code. This allows the user to focus on the column of interest and ignore all other paths, rather than stumble over intermittent irrelevant 'else' or 'case' statements. From the IDE's perspective, given the required parameters a single path of execution could be explicitly highlighted through a section of code. This may play a role in debugging.

Abstract different languages to a common representation.

Different languages express shared concepts using different words. But the overlap is not complete, as each language also expresses unique concepts. A representation could be created to act as a superset, encompassing all of the variations in the target set of languages. Each language would then implement a part of the common representation. Where languages overlap, the same abstract representation would be used.

To achieve this textually would require either changing the words and symbols of some languages to conform to the common representation, or write multiple versions of the common representation each catering for the words, symbols and conventions of a target language.

A more practical approach would be to use a superset of graphical symbols, of which each language implements a portion. Different descriptions of structures could be replaced with diagrams of

structures. This requires learning what the diagrams mean. This should not be as much of a burden as learning the meaning of new textual descriptions, especially if the diagrams map directly onto structures in the target language.

The diagrammatic approach has been used before, for example the Unified Modelling Language, UML.

Service database style queries.

The study of ontologies has already yielded a description format that allows database style queries of data. By describing a code base in an ontology language and using a reasoner, it should be possible to enable in-depth and ad hoc analysis of a project's code. Mapping between object-oriented languages and description languages has been investigated before [1, 2].

Display or allow access to the intermediate code generated from macros or annotations.

Macros and annotations acts as a powerful kind of meta-code. These allow the instructions to be passed to the pre-processor to automatically generate code according to a certain pattern. But as the programmer does not have access to the generated code before it is passed to the compiler, it may be difficult to debug or tweak. Allowing the user to generate code externally in the same fashion may help solve these issues.

Give special facilities for patterns or API's.

Patterns and API's are widely used. It would be beneficial to make them easier to use. Offering built-in or customisable structures to aid and visualise their use could benefit many projects.

Interactive testing.

Unit testing is both a widely endorsed, very useful and high maintenance doctrine. To make unit testing easier to use and debug, it would be useful to interactively visualise the flow of data through a routine.

Shift of emphasis.

Popular IDE's such as Visual Studio (Figure 1), CodeBlocks, Eclipse, Netbeans, JCreator, JGrasp, Lazarus and Delphi all follow the same approach. This is to present a dominant central text editor, with some tools around the periphery to show the structure or contents of the project files.

A shift of emphasis away from the text editor and towards the project structure may be beneficial. As the structure evolves, the user may edit the details, rearrange and refactor the project to account for newly discovered twists, issues or complications. This takes a top-down approach, focussing primarily on the framework rather than the underlying text.

Limit displayed information to currently relevant details.

To reduce the amount of concentration required, information not currently important to the programmer should be hidden or abbreviated, so as not to distract attention. For example, while editing a method, it should not (usually) be necessary to see the implementation of other methods.

But this is the case when editing a text file. By shifting focus to the structure of a project, it is possible to rather show the implementation of only selected methods.

Use of space.

Text editors generally use a single column of text next to the left hand margin, with some longer sentences spilling over towards the right hand margin. But screens offer a two dimensional workspace. Representations should make better use of the space available.

4. Aims

The project hopes to make the following claims:

1 Visual Programming

1.1 Languages may be abstracted to common diagrammatic representations.

2 Visual Programming/Human Computer Interaction

2.1 A visual interface will allow faster production, editing and understanding than ae traditional text editor.

3 Human Computer Interaction

3.1 The presented top-down approach provides faster high level editing than a traditional text editor.

3.2 Navigation developed and used in strategy games, such as the minimap, can be successfully and usefully applied to applications in other fields.

4 Artificial Intelligence

4.1 An ontology language may be used to successfully describe and analyse programming languages.

3. Design

Parser (Priority: Low)

A parser will be required to analyse the code and represent it in an intermediate format.

Intermediate format (Priority: Medium)

This intermediate format should facilitate analysis of the code, so that the interface can gather the information it needs to create suitable representations. To allow interaction, the format should be to accept changes and be able to communicate the changes back into textual code.

Interface (Priority: High)

An interface will be required to create representations from the information gathered supplied by the intermediate format. The interface must allow the user to interact with these representations.

At a minimum, the interface should represent classes down to the member variables and method definitions. The method declarations will be presented as text. The interface should present comments in a clean, logical, non-distracting way.

Other language features, especially for languages that are not purely Object-Oriented, may be added as the project progresses. These include features such as records (or structs), global functions and variables, interfaces, enumerations, macros and directives. A subset of method declarations may be represented graphically, for example using UML, to demonstrate the abstraction mentioned above.

Evaluation (Priority: High)

2.1 Faster production, editing and understanding

> Contrast the time taken to produce, edit and understand code using the diagrammatic interface and a traditional text editor.

3.1 Faster high level editing using top-down approach

> Contrast the time taken to compose and edit the structural outline of a project using the diagrammatic interface and a traditional text editor.

3.2 Success of strategy game interface

> Use an opinion survey.

4. Implementation

Parser

A parser has been written and is almost complete. If time allows it will be completed, but can be used in the current state to extract the required information.

Intermediate format

If possible, the intermediate format will be in a description language. Currently the Web Ontology Language (OWL), as used in [1] and [2], is being investigated. A number of reasoners are available. The open source (LGPL) reasoner Pellet is being investigated.

If use of a description language is not possible, the information will be stored in custom objects. This will require handwritten functions to iterate through and gather the required information required by the interface.

Interface

The interface will be written using the Pascal IDE, Lazarus. Lazarus provides powerful GUI support and is portable and open source. A mock interface has been written to demonstrate approaches to some of the ideas and issues raised above.

Evaluation

2.1 Faster production, editing and understanding

Record time taken to produce a small project, understand a pre-written project, and make changes to either of the above, using the project interface and an IDE of the tester's choice. The IDE should be equipped with the traditional central text editor. But the testers may use any tools or wizards available as the project interface claims to be better suited to this task than other available tools. A survey could be taken afterwards for user feedback.

Issues: Users will need to be familiar with the project interface beforehand, to avoid bias. The same project cannot be used twice for the understanding test, so two projects must be provided. Overall, understanding of both projects should be tested using both environments. To achieve this, testers should be split into two groups. For the production and editing tests, the testers may either practice beforehand using both environments, or be given different tasks for each environment.

3.1 Faster high level editing using top-down approach

Give testers some class hierarchies with method stubs to be implemented. Record the time taken, using project interface and IDE of their choice. A survey could be taken afterwards for user feedback.

Issues: The second time the testers implement the hierarchies, they should complete it more quickly than the first due to practice. Three options are presented to avoid this. The first option is that testers are split into two groups, one of which uses the project interface first and the other the IDE first. But there may not be many testers. The second option is that the two class hierarchies are different. But one may be easier to produce than the other. The third option is to allow testers to practice implementing the hierarchies before the test. This option is preferable, as it will avoid bias due to familiarity with their chosen IDE and unfamiliarity with the project interface.

3.2 Success of strategy game interface

Survey testers using an opinion scale from one to five.

Issues: There will probably not be many testers to survey.

References

[1] Kalyanpur, A., Pastor, D., Battle, S. & Padget, J. 2004. Automatic mapping of OWL ontologies into Java. In 'Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE)'.

[2] Koide, S., Aasman, J. & Haflich, S. 2005. OWL vs. Object Oriented Programming. In 'International Workshop on Semantic Web Enabled Software Engineering (SWESE)'.

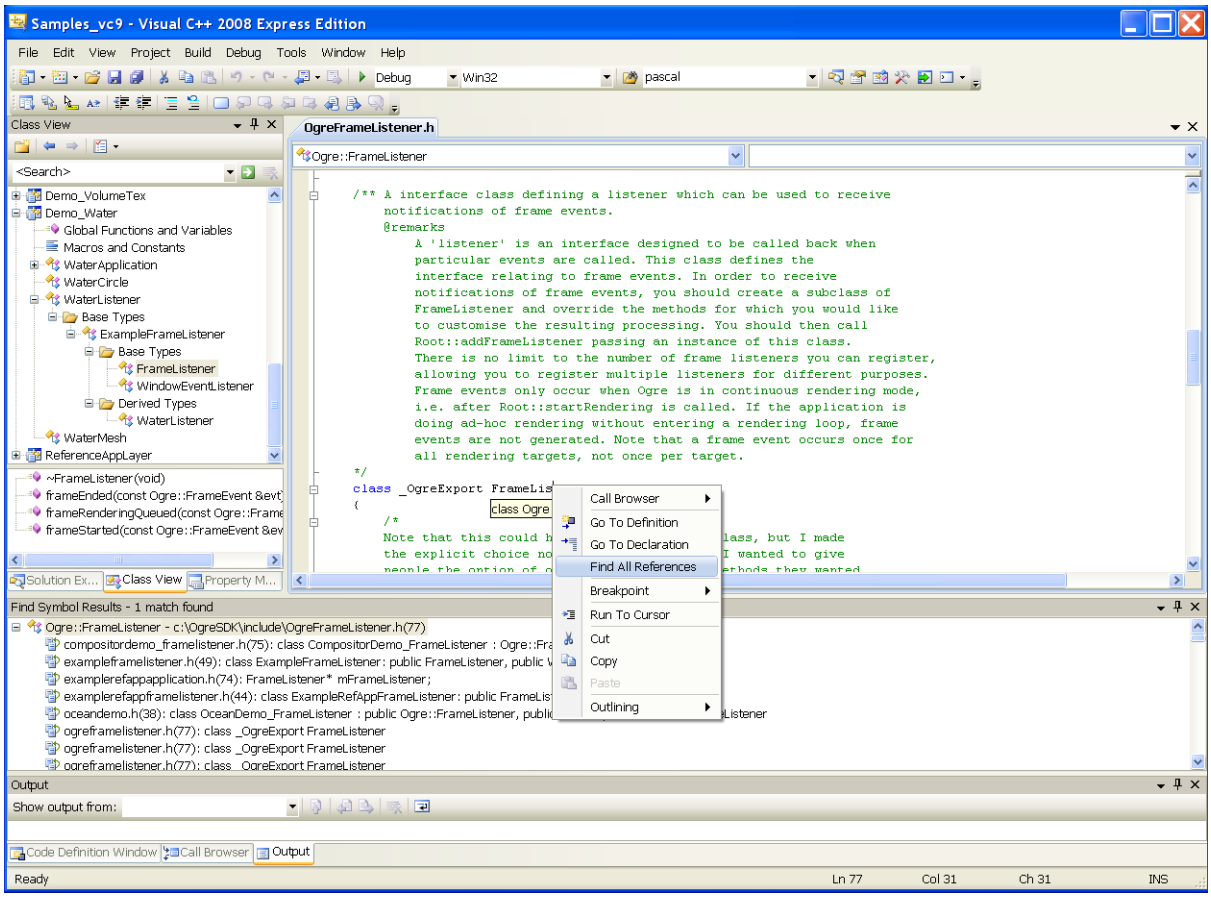

Figure 1: An example from Visual Studio. Projects are enumerated to make navigation easier, as are classes, global functions and data per project. The user may jump between definition and declaration, and search for references to an identifier. Popup boxes give extra information when the mouse hovers over a section of code. Note how, paradoxically, the bulk of good inline documentation breaks the flow of code making it difficult to study. Some tools, such as FPDoc, get around this by allowing the user to place comments in a separate file, which is merged when documentation is generated. This poses the issue of maintaining and updating multiple sets of files, although integrated tools may help.