



Brandenburg University of
Technology



Chair of programming languages and
compiler construction

Diploma thesis

Designing, implementing and integrating a structured C# code editor

Kirill Osenkov

kirill@osenkov.com

Advisor: Prof. Dr. rer. nat. habil. Peter Bachmann

Co-Examiner: Prof. Dr. rer. nat. Claus Lewerentz

CR-Classification: D.2.3, D.2.6

June 1, 2007

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig angefertigt habe. Die verwendeten Hilfsmittel und Quellen sind im Literaturverzeichnis vollständig aufgeführt. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Eingetragene Warenzeichen und Copyrights werden anerkannt, auch wenn sie nicht explizit gekennzeichnet sind.

1. Juni 2007

Cottbus

Abstract

Programs are represented as text by most code editors. Structured editors, on the contrary, directly display the parse tree of a program as a hierarchy of embedded blocks. This way the visual layout illustrates the structure of the program and allows for atomic operations on language constructs. The structured editor lets developers avoid syntax errors and concentrate on the meaning of the program instead of formatting.

In the scope of this thesis, an experimental structured editor is created for a subset of C# 1.0. It is integrated into the SharpDevelop IDE as an add-in and supports language-aware code completion. The implementation of the editor (.NET) is based on a framework specifically designed for building structured editors. The framework, the editor and the integration add-in are all documented together with the architecture and implementation details.

Zusammenfassung

Programme werden in den meisten Quelltexteditoren als Text repräsentiert. Strukturierte Editoren stellen dagegen den Syntaxbaum des Programms visuell als eine Hierarchie der geschachtelten Blöcke dar. Das Layout hilft dabei, die Struktur des Programms besser zu visualisieren und atomare Operationen bequem durchzuführen. Ein strukturierter Editor hilft Syntaxfehler zu vermeiden und lässt den Entwickler auf dem Inhalt statt Formattierung zu konzentrieren.

In Rahmen dieser Diplomarbeit wird ein experimenteller strukturierter Editor für eine Teilmenge von C# 1.0 entworfen und implementiert. Der Editor ist mit der SharpDevelop Entwicklungsumgebung integriert und bietet kontextbasierte Code-Vervollständigung. Die Implementierung des Editors (.NET) beruht auf einem generischen Framework zur Entwicklung von strukturierten Editoren. Das Framework, der Editor und die Integration mit der Entwicklungsumgebung sowie die Architektur und Implementierungsdetails werden in dieser Arbeit dokumentiert.

Acknowledgements

I would like to thank Professor Dr. Peter Bachmann for initiating and supporting this research. It was an enjoyment to work with such a good advisor on such an interesting project. I'd also like to thank Professor Dr. Claus Lewerentz for being a co-examiner of the thesis. Many thanks go to fellow students and friends with whom we worked together on the structured editor framework.

I'm endlessly grateful to my wonderful mom. Thank you for your love and for always being there for me!

Last, but not least, I thank my amazing girlfriend Vlada, for inspiring me and making me happy!

Kirill Osenkov

June 1, 2007

Cottbus, Germany

Contents:

1. INTRODUCTION	7
1.1. TEXT VS. STRUCTURED EDITORS	7
1.2. INTEGRATION OF AN EDITOR WITH THE IDE.....	8
1.3. PRINCIPLES OF PLAIN TEXT EDITORS	9
1.4. PRINCIPLES OF STRUCTURED EDITORS.....	17
1.5. DISADVANTAGES AND POSSIBLE DIFFICULTIES	25
2. EXISTING RESEARCH.....	29
2.1. HISTORY	29
2.2. INTENTIONAL PROGRAMMING	31
2.3. JETBRAINS MPS	32
2.4. THE SYNTHESIZER GENERATOR.....	33
2.5. OTHER IMPLEMENTATIONS.....	33
2.6. SUMMARY	34
3. FUNCTIONALITY OF THE STRUCTURED EDITOR.....	35
3.1. CREATING A PROGRAM	36
3.2. NAMESPACE MEMBER DECLARATIONS	39
3.3. TYPE DECLARATIONS	41
3.4. ACCESS MODIFIERS	42
3.5. CLASS AND STRUCT MEMBERS	44
3.6. STATEMENTS.....	47
3.7. CONTROL STRUCTURES	48
3.8. COMMENTS	49
3.9. CODE COMPLETION	50
4. ARCHITECTURE.....	51
4.1. THE EDITOR FRAMEWORK	51
4.2. UTILS	53
4.3. CANVAS	54
4.4. CONTROLS	54
4.5. CORE.....	55
4.6. IMPLEMENTATION OF THE C# EDITOR	56
5. BLOCKS.....	58
5.1. DATA STRUCTURE	58
5.2. TREE ORGANIZATION	60
5.3. OPERATIONS ON THE DATA STRUCTURE	62
5.4. CONTAINERBLOCK	63
5.5. ROOTBLOCK	65

5.6. HCONTAINERBLOCK, VCONTAINERBLOCK	65
5.7. LINEARCONTAINERBLOCK	66
5.8. TEXTBOXBLOCK	67
5.9. TEXTBOXBLOCKWITHCOMPLETION	68
5.10. LABELBLOCK	69
5.11. UNIVERSALBLOCK	70
5.12. BUTTONBLOCK	70
5.13. EMPTYBLOCK	71
6. IMPLEMENTATION OF THE FUNCTIONALITY	74
6.1. DESIGNING THE USER INTERFACE	74
6.2. FOCUS	75
6.3. EVENTS AND USER INTERACTION	78
6.4. ACTIONS	81
6.5. CONTROLS	87
6.6. VISIBILITYCONDITIONS	89
7. IMPLEMENTING THE C# EDITOR	91
7.1. THE PROJECT STRUCTURE	91
7.2. DEFINING DATA STRUCTURES (BLOCKS)	92
7.3. DYNAMIC HELP	100
7.4. LANGUAGESERVICE	100
7.5. CLASSNAVIGATOR ALGORITHMS	101
7.6. STAND-ALONE EDITOR WINDOW	103
8. INTEGRATING INTO SHARPDEVELOP	105
8.1. THE SHARPDEVELOP IDE	105
8.2. ARCHITECTURE	106
8.3. SHARPDEVELOP ADD-IN	110
8.4. ROUND-TRIPPING	113
8.5. IMPLEMENTATION OF THE ADD-IN	115
8.6. CODE COMPLETION	119
9. SUMMARY	122
9.1. FUTURE RESEARCH DIRECTIONS	123
9.2. DRAWBACKS OF THE CURRENT IMPLEMENTATION	125
10. REFERENCES	127
11. LIST OF FIGURES	131

1. Introduction

1.1. Text vs. structured editors

Most developers currently use text-based editors to edit source code. These editors are highly specialized and integrated into an IDE¹ to provide a comfortable editing experience. An IDE typically provides many helpful features such as background compiling, code completion, snippets, navigation, refactoring, debugging etc.

But at the ground level, to edit the program the developer still deals with characters and lines of text. The hierarchical structure of a program is represented with the help of specialized markup tokens and indentation to delimit language constructs. For example, C-family languages use the curly braces { and }, XML uses tag pairs <X> and </X> and VB.NET uses `Keyword – End Keyword` pairs.

Internally a program is represented by a hierarchical structure called a parse tree or an AST². It is being recovered from the source text using a scanner and a parser. The AST can be visually displayed using embedded blocks. A *structured editor* (also *structure editor*) allows the user to interact with the syntax tree directly by interacting with these blocks.

¹ IDE = Integrated Development Environment

² AST = Abstract Syntax Tree

Language constructs are the new editor “atoms”, in contrast to characters and lines of text. Thus a round-tripping from text to AST and back is unnecessary.

Note on terminology: some empiric web-research has revealed that the term „*structure editor*“ is more used in the context of chemistry and biology to denote editors for molecular and cell structures. On the contrary, the term „*structured editor*“ is more often used in the context of editing documents (for example, in [Amaya]) and that is why it is preferredly used in this thesis.

We’ll use the term *plain text editors*, *traditional editors* or simply *text editors* to denote the currently widespread approach of editing program as text.

Such a tree-based representation can provide many advantages. The advantages are two-fold: advantages for the user of the editor (more editing comfort and improved usability through language-awareness), as well as advantages for the developers of the IDE (more consistent, robust and extensible architecture through model-view-controller approach).

Wesner Moise envisions in [WesnerM1] 2004 the development of a structured alternative to plain text editors:

“Text editors are going to go away (for source code, that is)! Don't get me wrong, source code will still be in text files. However, future code editors will parse the code directly from the text file and will be displayed in a concise, graphical and nicely presented view with each element in the view representing a parse tree node.”

However, this thesis supports a vision where structured editors won’t fully replace text editors, but rather complement text editors as yet another view on the same internal code model, to supplement richer and more intelligent editing experience where possible. It is important to note that the language underneath remains the same: it’s all about a new representation of same old language constructs.

1.2. Integration of an editor with the IDE

We’ll talk about editors in the context of a typical IDE. For this purpose let’s divide the architecture of a typical (not every!) IDE into logical layers (tiers):

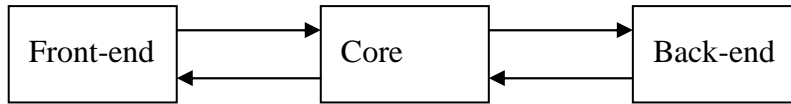


Figure 1 – layered architecture of an IDE

Let’s call the part which is closest to the user a *front-end* – the user interface, the editor surface and the documents. A front-end is usually built upon some application framework – a complex UI library which provides hosting with UI elements (dock panels, menus, toolbars, document windows, etc).

Although the *back-end* normally consists of many different components, the main part of it is of course the compiler. A back-end also usually includes a debugger, a storage system (file-based or version control) etc.

For our purposes, we’ll call the mediating part the *core*. The core binds together all parts of the IDE, in particular, it connects the compiler to the user interface. An important part of the core is the *language service*, which allows the IDE to “understand” the code and which makes the IDE “intelligent”: the DOM¹, the parser, the code colorizer, the resolver, the background compiler, the project system etc. The core makes the text editor *language-aware*, because the feedback provided to the user is based on the syntax and semantics of the programming language as well as knowledge about the program being edited.

1.3. Principles of plain text editors

1.3.1. Text data structure

The main abstraction behind a text editor is a stream of characters or a sequence of text lines. This abstraction is widely used in the front-end and in the core to work with programs as text files. The API interface of the editor control only works with text and does not “know” about the AST.

¹ DOM = Document Object Model

In the core, two principal data structures are intertwined together: the text data structure and the AST. The AST is mostly being used by the language service.

Interestingly enough, although the core “knows” about the AST, the interface between the core and the back-end is still based on text in many IDEs. The back-end (e.g. a compiler) acts as a black-box:

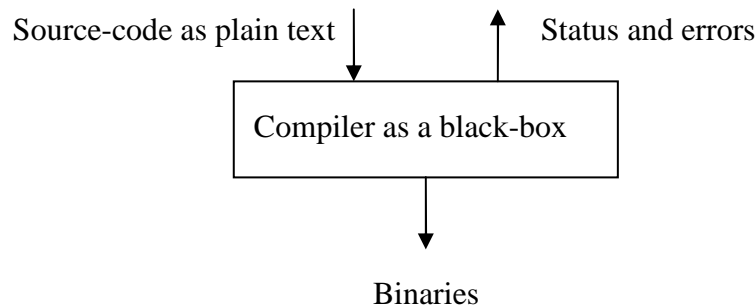


Figure 2 - compiler as a black-box for the IDE

The interface of the compiler doesn’t provide “knowledge” about the internal AST data structure: plain text goes in, is internally parsed into a private AST and binaries are generated.

This black-box isn’t extensible. There is no possibility for the user to intercept the process of compilation and to gain access to the internal AST of the compiler. There are a number of scenarios, where a dramatic increase of possibilities in the direction of meta-programming and generative programming would be possible, if only one could have access and work with the compiler’s data structures during the compilation. One could use it for intelligent preprocessing, custom language extensions, code injection, AOP¹ etc. Vendors that provide IDE extensions are often forced to rewrite huge parts of the compiler functionality which otherwise could have been exposed as a clean compiler API operating on the “live” AST structure.

It is also characteristic to all text editors is that the text editor acts like a black-box as well. The current program is being presented to the user, and the user is carrying out actions, which the core is totally unaware of. The meaning of the user’s changes is being completely lost at the very moment when the change is made and only reconstructed with a

¹ AOP = Aspect Oriented Programming

lot of effort from the background compiler. Basically, the core has to re-parse with every user edit, because it has no knowledge about the intent (the meaning) of the user change. The user could even replace the entire program text in a single action, and only a full re-parse can reconstruct the knowledge about the program.

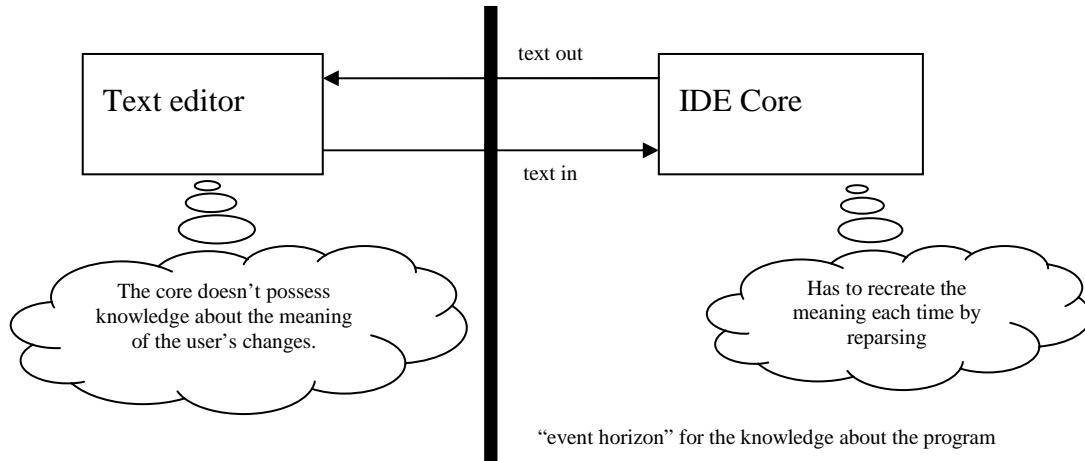


Figure 3 – the editor as a black-box for the IDE

This is the “black hole” of plain text – anything can happen out there and the parser doesn’t have any idea. The IDE doesn’t know what happens to the code between the parsings. This knowledge is lost and found every time. The reparse algorithms can get incredibly complex, when an incremental background compiler is used. In this case, changes are tracked artificially, by recalculating delta differences each time between two known text states or by differentiating “destructive” vs. “harmless” edits (if a reparse is necessary).

It is noteworthy, however, that the decoupling and encapsulation of the text editor and the compiler from the IDE might provide certain flexibility with regard to the implementation – one could change the language (the compiler) without changing the interface that binds the compiler to the outside world. However, as soon as one needs to expose compiler functionality to the rest of the IDE (for example, for the purposes of the language service – intelligent code completion etc.), this observation is not valid anymore – the implementation of the language service has to be kept in sync with all changes to the compiler, regardless if the interface to the compiler changes or not.

1.3.2. Advantages of text editors

The “black-box nature” is at the same time an important advantage of all text editors – it gives the user full flexibility and freedom to make absolutely any changes to the text, in any order, not bound by any semantical constraints. This allows for intermediate editing states where the program is incorrect. In many cases it is easier to bring the program into a temporary incorrect state to reach the desired correct state. A prominent example for incorrect intermediate states is transforming

```
int i = 0;  
i = 42;
```

into

```
int i = 42;
```

by placing the caret at the beginning of the second line and pressing [Backspace] several times. At some point the code snippet will look like:

```
int i = 0i = 42;
```

which is incorrect, but fully OK as long as this code remains within the editor.

As the editor control is decoupled from the rest of the IDE using a generic language-agnostic API surface, it becomes easy to implement, generic, flexible and almost universally usable.

Text editors allow for editing programs pretty fast, too. Editing speed is an important advantage of text mode, to which programmers are used. It is crucial to preserve this advantage to let users benefit from this.

Another implicit advantage of text editors is familiarity. Text editors are something all developers use all the time during coding, and they got used to it very well. For a programmer, there is basically no need to learn how to use the editor, once a new language or environment comes out. Moreover, text editors are actually very effective, so many developers are actually totally pleased with the editing experience and don’t have any complaints. This, and the fact that text editors are so widespread, will probably be the reason, why structured editors never fully replace text editors.

1.3.3. Usability problems of text editors

However the flexibility of text editors comes at a price – the users have to take care of the syntax and formatting. They have to help the editor to convert the program into an intermediate representation by manually separating language constructs with separators like ‘{’, ‘;’, ‘(’, ‘//’. Even if the IDE provides automatic code formatting (employing the pretty-printer from the language service) and code snippets to automatically insert constructs like { }, the user is still involved with manually inserting tabs, spaces, blank lines, semicolons etc.

It makes sense to compare usability of text editors to that of structured editors by measuring and comparing the number of keystrokes (atomic user input actions) required to achieve the same task.

For example, let’s consider creating an empty statement block for a method in Microsoft Visual C# 2005:

```
void Foo()  
{  
  
}
```

It takes three to four keystrokes to insert two blank lines and to position the cursor on the first line (either [Ctrl+Enter] twice or [Enter], [Enter], [UpArrow], [UpArrow]). Then it takes three keystrokes to type in “void “ (IntelliSense™ completes “vo” to “void“ when [Space] is pressed) and another five for “Foo()”. And then it takes 6 keystrokes to insert the curly braces and to position the caret in between with the right indentation. One could accomplish the same with just 8 keystrokes instead of 18, which might seem not a big difference at first, but it definitely gives an overall improvement of editing experience.

It should be impossible for the caret to enter the indentation space to the left of the blocks. If the code is properly formatted (and this always should be the case), there is no need to edit anything to the left of the first significant character in the line. Now it is possible to penetrate the indentation space by pressing the [Left] or [Home] key. All the tabulation should be done automatically, and hence the need to penetrate the left tabulation margin should be eliminated.

However, experienced programmers are used to typing so fast, that this doesn't bother them at all. Besides, some languages (for example VB.NET 2005) take more care of typing, completion and formatting (code snippets, auto-format on paste, etc), which reduces typing efforts to a minimum. So it is a common belief that the usability of traditional text editors is not an issue, especially in presence of such enhancement tools as, for instance, JetBrains ReSharper ([ReSharper]) or Whole Tomato Visual Assist X ([VAssist]). However for beginners, a learning curve to use an editor effectively is still pretty steep, so formatting and the necessity to take care of the syntax is an issue.

1.3.4. Implementation difficulties with text editors

The black-box nature of the editor and the compiler dictates some implementation peculiarities when developing an IDE. One of the biggest ones is the complexity of the round-tripping between text and AST. The one direction is more or less straightforward – pretty-printing, auto-formatting and code generation. The other one is the tricky one – going from code to the AST. This complexity is classically being tackled by the parser – in the sense of the famous dragon book ([AhoSeUl]). However what the successes of the compiler science don't currently fully cover is the implementation of the language service: “understanding” code and providing intelligent feedback about it. The typical problem is implementing the expression finder and the resolver – given a text stream and the caret position, it is required to reconstruct the language construct under cursor and to provide user feedback about it (code completion, method info, parameter info or even colorizing). This task requires finding the current context (find the class and method that currently contain the caret, as well as the current expression), reparsing the necessary text, updating the internal representation and actually performing the task.

The implementation complexity of round-tripping is due to the fact that most IDEs currently are built around the text data structures, and not around the AST.

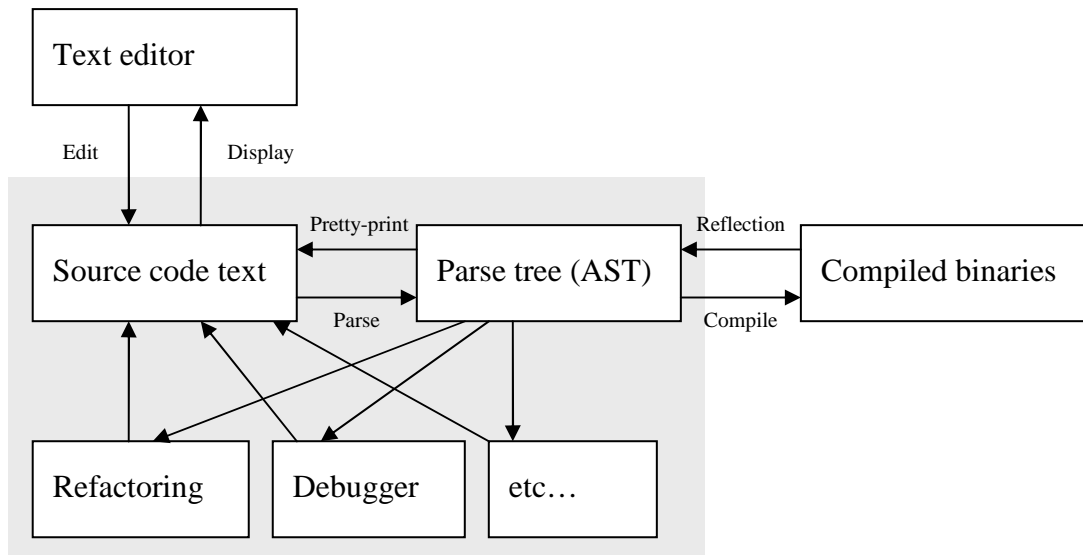


Figure 4 – round-tripping between the AST and IDE components through code

To avoid this complexity, an IDE could be built around precisely defined, language-aware and observable syntax trees, which would serve as the Model in the MVC¹ triple mentioned in [GoF]. A text editor should be just a View, a thin presentation layer which maps user edits to the AST using a controller (a set of hierarchical controls representing language constructs). All other IDE components should only deal with the AST (also often called DOM, CodeDOM, Code Model, Intermediate Representation, Parse Trees, etc.).

Like a database management system guarantees the fulfilment of ACID principles, the IDE should guarantee the integrity of its data structures. The reasons why people invented DBMS to replace plain text are often the same for the source code (see [SCID]).

A compiler shouldn't be a black-box, but a clean API surface instead, which exposes methods to transparently operate on the AST and to transform it, thus making the compile functionality reusable and extensible (pluggable). In such a way one could easily plug-in custom transformations or code generations between the parser and the code generator. Authors of IDE extensions could thus reuse the compiler functionality, without the need for own third-party parser, resolver, etc.

¹ MVC = Model-View-Controller

A debugger could map to language constructs instead of text positions in code, thus preventing that text positions can get out of sync. Thus many known bugs with line and column number offsets could be easily prevented.

This idea brings us to the possible approach of using structured editors to directly operate on the AST:

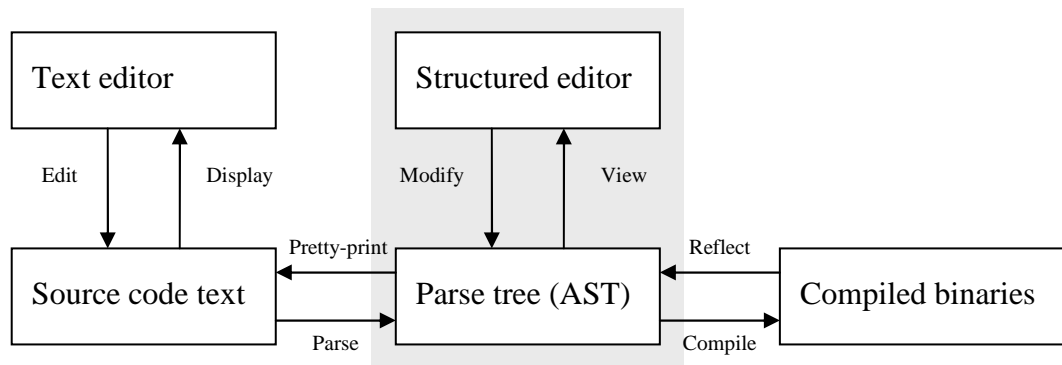


Figure 5 – a structured editor directly operates on the AST

Shifting attention to the AST instead of text data structure would allow bypassing the round-tripping step:

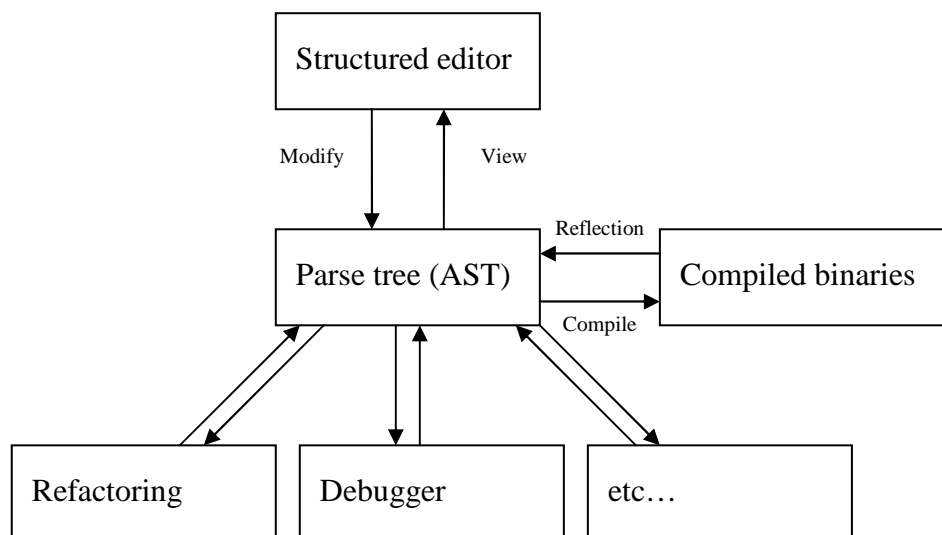


Figure 6 - a possible architecture of an IDE built around the AST

It is important to understand that an architecture of an IDE built around a structured editor and syntax trees doesn't necessarily imply that the source code has to be stored in a different format, perhaps in a database. The source code could still be stored in usual files.

The file format of programs doesn't even need to change. A parser could load the AST from source, and a pretty-printer could save the AST back into the source code files. The editor could even preserve user's formatting when saving.

1.4. Principles of structured editors

In contrast to the modern text editors which could be called *language-aware*, you might call structured editors *language-driven*. This expresses a more strict and constrained approach to editing, where each change is carried out on language constructs and not on a list of text lines. The syntax and grammar of the language define what language construct can be inserted at the current insertion point, and the editor provides choice for the user.

1.4.1. Atomicity

Unlike plain text editors, a structured editor cannot represent or operate on a part of a language construct – it operates on AST elements as a whole. For instance, in plain text you can have an incomplete construct with an opening brace but without the closing brace. A structured editor doesn't need braces at all. In text, you can delete part of a concept by deleting incomplete selection. In a structured editor, you can only delete a whole concept.

The atomicity of editor operations guarantees that language constructs are being added, moved, renamed or removed as a whole. The typedness of language constructs would only allow correct embedding of concepts – each concept can only host those concepts which are allowed by the language grammar. For example, you wouldn't be able to move a method to a namespace level, or a foreach statement to a class, or declare an implementation on an interface method.

Each change to the data structure is of transactional nature and is language-aware. For instance, one would speak of changes in terms of “a class was inserted” instead of “text was inserted”. It is easier to track all the changes in an Undo/Redo buffer and to compose multiple changes to a transaction.

1.4.2. Correctness

Most syntax errors are simply impossible within a structured editor, so the program being edited is always syntactically correct (with minor exceptions). For example, representing hierarchical concepts as nested blocks (and not with a set of paired delimiters) guarantees

that the curly braces and #region directives will always be balanced. Curly braces earlier used to delimit tree nodes become now unnecessary.

When you are inside a block, only blocks of certain type can be inserted, dependent on language syntax. You can continuously move your caret through your code as if it were plain text. All the blocks are formatted automatically.

Few remaining errors, which are not caught by the structure itself, could be easily found and highlighted by using a tree checker visitor. The implementation would be simpler and wouldn't theoretically even require a scanner or a parser for this purpose. Thus the editor can automatically ensure syntax correctness by its language-awareness.

1.4.3. Usability

Since a structured editor has more ready knowledge about the program being edited and the semantics of the language constructs, it can more easily offer more intelligent user feedback on the program and offer only valid choices based on the current context. The context itself can only be a correct language construct or a correct insertion point where a language construct can be created.

A structured editor can also reduce the number of required primitive user input operations (key presses, mouse clicks and moves). For example, it is sufficient to select a block and hit the [Delete] key to remove a concept. In a text editor, one would first have to select the text which corresponds to the concept and only after that press [Delete]. When a concept spans more than one screen (for example, a big class), it is still easy to select and delete it as a whole in the structured editor, whereas selecting the class in a text editor would require scrolling.

Each block can be selected, collapsed, commented, cut or copied to clipboard, moved into another position in the parent block or into another block with just one click or key press. Entire block hierarchies are selectable just like usual blocks.

With a structured editor the user doesn't have to worry about indentation, semicolons, curly braces etc. The editor takes care of the syntax so that the user can concentrate on editing the program and not on remembering the syntax for a language construct. The code is being automatically "formatted" as it is typed in, because it is actually displayed from a live AST, which is always correctly "formatted". The user mostly enters important content and not irrelevant formatting symbols.

Structured editors actually open wide possibilities for usability research. It is a complex task to gather statistical information about how different developers perform elementary typing operations and what elementary user input actions are required (mouse and key clicks, mouse moves, pen gestures). One thing is sure: developers don't actually notice, how exactly they type – they just type without thinking about it and develop some sort of reflexes when working with an editor for a long time. Structured editors could substantially shorten required user input actions when compared to text editors. This will most probably be a problem for those who are already using text editors for a long time, but newcomers will probably appreciate it, especially because they don't really have to remember the syntax (“were there semicolons inside a `for` loop or commas?”).

Context help for an editor could provide important hints for the user based on the current selection and caret position. Implementing context help would be much simpler too – one could simply declaratively add help pages to the syntax constructs, without the need to parse the source code to determine what construct is currently near the caret.

An intelligent scrollbar could allow easy and powerful code navigation. The scrollbar could contain miniature portions of the code document. This approach is being explored by the Human Interactions in Programming group [HIP] inside Microsoft Research – in a project called “Code Thumbnails”.

A so-called “breadcrumb” control could be provided for improved navigation experience. When a block is active (focused), an extra control could show a horizontal list of all parent blocks as a chain. Each link of the chain could actually be a hyperlink to the corresponding parent block, or a combo-box with the drop-down list of all other siblings at this parenting level.

Another possible future direction for usability research is key stroke prediction based on statistics – gathering and accumulating typing experience from users all around the world could provide some AI-driven preselection what construct is going to be used next, just like IntelliSense in Visual Studio 2005 remembers the last used type and member.

For those who are sceptical about the usability of a structured editor because it will be so unusual and difficult to use, Wesner Moise gives a probable answer to this in [WesnerM1]:

. ... Such graphical editors won't actually appear much different from text editors; they may even look the same as text editors, with some differences.”

Thus structured editors could mimic traditional text editors to provide familiarity but still retain the advantages of structured editing.

1.4.4. Presentation

Another good thing about structured editors is that they are very easily skinnable, because the content is decoupled from the presentation. Just like a CSS stylesheet can provide a totally different look for an XHTML document, one could easily change the presentation of how language constructs are visualized on the screen. It is the content and the structure of that content that matters.

These are the reasons why XHTML + CSS were developed to replace old HTML where the presentation was mixed with the content. Another example for this development are styles in word processors, which are definitely better than inline formatting scattered all over the document in multiple places.

When the presentation can be adjusted independently of content, preserving team's coding guidelines becomes a snap. One could virtually "edit" the teams coding guidelines in a special domain-specific editor, for example in a graphical design software with gradients and visual effects, and the entire source code would be automatically displayed with this style on the fly. Or, every developer could even have own coding guidelines (style library) – and have all the code automatically displayed with her/his favorite style.

When you open the program in another IDE with different settings, the code could be displayed using these settings automatically, without the need to manually reformat it. Reformatting code would become unnecessary. Code could be automatically formatted for printing based on one of printing templates.

Again, this is another manifestation of the MVC principle: one could easily switch views or use several different synchronized views at the same time, as long as the model (content) is clearly separated from the presentation.

Another nice feature of structured editors is the possibility to collapse everything, to only show the currently relevant code. This could go much further than by text editors – one could even apply custom sort or filter to display only relevant things in the desired order. The order in which the declarations are physically stored in the file would become separated from the logical order in which they are presented to the user. Different developers could even have the same code sorted as they want it.

This is a common reason for disagreement in teams about coding guidelines – some prefer to have all the fields at the beginning of the class, and some prefer to have each backing field attached to its corresponding wrapping property/getter/setter.

Structured editors are also well suited for embedding other mini-languages (DSL¹) within the main (host) language. Just as LINQ features are a sort of “embedded SQL/XML” in C# 3.0, structured editors could easily embed custom DSL, for example mathematical notation for expressions, with the square root sign and vertical fraction notation.

An editor could contain custom interactive controls to improve the editing experience for specialized content, for example a color-choice popup box where a variable of type Color is required.

Other visual formatting could be added as necessary. For example, custom lines, rules and delimiters could be used instead of ASCII-art style comments:

```
// =====
```

Lutz Roeder gives insightful examples of such enrichments in [LutzR1].

1.4.5. Simplification

A structured editor could simplify and automate many programmer’s tasks that now have to be carried out manually. Backing fields for properties can be entered and displayed concisely (C# 3.0 now does this with its auto-implemented properties).

1.4.6. Extensibility

A structured editor could provide a platform for extending the language by defining some equivalent of macros. One could define a new type of a language construct externally and then just plug this type into the grammar of the editor. Thus very concise and expressive shortcuts for long pieces of code could be defined, which could be expanded into longer code sections during the code generation. The difference between such macros and, say, C #define macros is that the structured macros (unlike C macros) would be language-aware, and thus all the type-checking and verification can actually take place. The problem

¹ DSL = Domain Specific Language

with C macros was exactly the lack of language-awareness – the substitution took place before parsing, at the preprocessing stage. With language-aware macros the substitution takes place after parsing – language constructs are being inserted into the existing AST, thus allowing all checking visitors to run on the final tree. Moreover, the editor’s grammar and type-checking facilities could only allow inserting macro instances at correct places in the tree. This eliminates the problems of text-based macros and preprocessing.

The [Nemerle] programming language gives a great example of language-aware macros in a text-based language.

The structured, factored nature of the editor would allow to more easily extend the language without actually extending the language. Plugging in new language concepts probably wouldn’t even require recompiling the editor. A new ecosystem for “language plug-ins” could arise, which would allow each software vendor to develop tools for the product parallel to the product itself, in the sense of Microsoft’s Software Factories initiative ([SoftFact]).

A great research about extensibility of structured editors is given by Intentional Programming ([IntentSoft], [Simonyi1]).

1.4.7. Implementation

An IDE built around AST structures and backed by a structured editor that directly operates on this structure could really simplify the architecture and implementation of many components, such as code completion, refactoring, navigation, debugger, edit-and-continue, etc. The thing is, it is easier to make the IDE intelligent if there is no need to constantly round-trip between text and AST. A good example is extracting a method, which would be simpler to implement in comparison with some current implementations, where a lot of effort has to be put into the round-tripping. When some code has to be inserted into the program with a text editor, a major problem is finding the correct place to insert and take care of the formatting. A structured editor takes care of the formatting.

An important advantage in the implementation is eliminating the need for a background compiler. When the AST is kept up-to-date by the editor itself, there is no need to reparse code in regular intervals in a background thread. When there is no background thread, there is no danger that the program information would get out of date, which can be the case with background compilation. Implementing a background compiler is not an easy

task and freeing the developers from the need to implement one could simplify the overall architecture of the IDE and keep it more robust and consistent.

1.4.8. Performance

It may well be the fact that the performance of a structured editor (especially when doing complex language-aware operations such as code completion or refactoring) would far surpass that of a text editor. Resolving and reparsing is a common bottleneck of traditional language services, and a structured approach could handle this elegantly.

1.4.9. Storage and version control

When the source code is stored as text, the internal representation isn't stored with it – it is being reconstructed every time when the source is loaded into the IDE. That's why the changes are expressed in terms of changed lines in version control systems. Moreover, current version control systems do not provide feedback about, for instance, how many classes were changed, how many methods have been added/removed/moved/edited, etc.

When the parsed and resolved AST is stored in a versioned and transactional database, changes could be registered in fine-granular, language-aware manner: classes added, methods edited, variable renamed, etc.

Thus the editor ("View") could reside on the client and the code being edited ("Model") could be inside a database repository on a (possibly remote) server. This could even potentially allow several people to edit the same program at the same time (while viewing updates in realtime), which is more fine-granular than current version control systems.

Branching and merging operations could be carried out more easily and in terms of changes to language constructs. Because the order of declarations on a class or namespace level is visible to the system, moving a method within the same class wouldn't even be considered a change for the end-user – the users who carry out forward- and reverse-integration processes now can concentrate on the meaning of the program, instead of formatting.

Another advantage of language-aware source code repositories that store AST in a database is a requirement, that every code being checked in must be at least syntactically correct. When implementing such a system, it surely would be possible to only allow to check in code that compiles, because the system would understand the code which is being checked in.

A language-aware version control system can prevent breaking builds.

Moreover, the check in process as such, could be replaced in the future by more fine-granular “transactions on code” – literally each change would be registered in the repository as a transaction and one could group changes manually to the desired granularity level, thus grouping changes to larger entities (for example, one day’s work). The change groups could be grouped too (design pattern `Composite`) to encapsulate features, milestones, products, etc.

A web-service based, programming language independent source code repository would allow developers to access shared code from desktop computers or mobile devices. The code can be automatically formatted and represented in any preferred way on the client (by the editor), whereas the repository is formatting-agnostic.

A lot of insight about storing source code in some intermediate representation (e.g. a relational database) is given in [SCID]. A big application of storing source code in database could be static analysis tools.

1.4.10. Static analysis and source code querying

Static analysis of source code and source code querying are becoming more and more popular nowadays. Tools like [FxCop], [NDepend], [Semmle] and [NStatic] either store the parsed code in a relational database, parse the code on the fly or analyze assemblies/byte-code. Either way, they operate on the AST, and if the AST exists during code editing and is always up-to-date even without the background compiler, such tools could greatly benefit by eliminating the need for a custom parser or even provide a real-time “as-you-type” analysis experience, where possible.

1.4.11. Help with learning the language

A good thing about structured editing is that it is more explicit about the language constructs being edited. A structured editor better reflects the inner structure of a program, how it is seen by a compiler.

Currently, the first thing beginner programmers learn is the textual syntax of a programming language, not the language constructs themselves. A common beginners question when learning programming is “what [language construct] can I actually insert at the current position”? Other beginners problems are, for example, remembering the right

keyword, spelling and order of arguments, typing and formatting, matching curly braces, indenting blocks with tabs etc.

These questions could be better answered by a structured editor, where syntax constructs and their embeddings can be more clearly visualized, and an alternative list would only show possible constructs.

Also, programmers use different programming languages. In particular, the Microsoft .NET Framework supports many languages in a single environment. Understanding code, written in another .NET language, is for many an obstacle. However it is often the case that the languages differ mostly by syntax and it is the syntax that prevents a C# developer to immediately recognize a constructor in “Sub New . . . End Sub”. The structured approach allows to at least partly erase the boundaries between different .NET programming languages and those who use them.

1.5. Disadvantages and possible difficulties

1.5.1. Usability

Probably the biggest predicted problem of a structured editor implementation is the usability problem. While editing text is straightforward and consistent (only some basic operations are needed, such as insert a character, delete a character, move the caret, etc), editing the AST structure on the screen requires learning how to operate on each node of the program. Moreover, editors that force the user to use menu, toolbars or the mouse during editing are most probably destined to fail. Text editors allow to edit programs using keyboard only and users won't give up this ability.

That is why, a structured editor must be more usable than a text editor by at least the amount necessary to convince users to transition to structured editing. This puts tremendous constraints on the user interface, the biggest of them being able to use keyboard only. Also, the comparative amount of key press operations to achieve same functionality must be lower than amount required in text editors. Another restriction is that the editor must be consistent (different language constructs must be operated the same way). Only under these strict circumstances the end-user will even consider giving the structured editor a chance.

1.5.2. Lack of familiarity

Even if structured editors *could* be better than text editors in all areas, there would still be an important advantage of text editors over structured editors: users throughout the world are well familiar to text editors. For example, if you have ever used Notepad, you will be immediately familiar with the editors of [SlickEdit], Visual Studio ([VS]), [ReSharper], [Eclipse], [IntelliJ] IDEA, etc. Structured editors must offer sufficient value over text editors while preserving all their advantages, and this is a very complicated problem.

1.5.3. Lack of flexibility

While text editors become more intelligent and usable by applying more and more constraints to what can be typed in plain text, structured editors move to the same goal from the opposite direction: they start with the strictest constraints that arise from the hierarchical nature of the program and relax some constraints to allow temporary incorrect program states for improved and simplified editing experience.

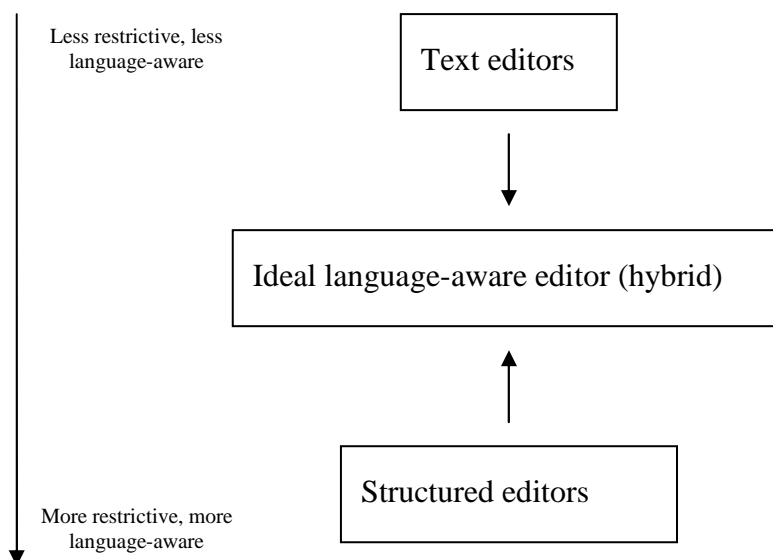


Figure 7 – approaching the ideal editor from different sides

As noted in section 1.3.2, text editors aren't limited by the semantics of the language and allow the program to be in an incorrect state, for example, if it is required to provide more flexible editing experience. Structured editors either lack this freedom to corrupt the program for the sake of usability, or must weaken (or sacrifice) the constraints posed by correctness checks to provide the user with the necessary flexibility. This is a compromise every developer of a structured editor must face sooner or later.

The editor shouldn't stand in the way. The user shouldn't notice the editor just like people don't notice the pen they're writing with.

Somewhere in between there is the editor of the future, with just the right amount of flexibility and language-awareness.

1.5.4. IDE dependency

Software developers and development teams choose to develop a product using a specific programming language or a combination of languages. This brings in a language dependency, but still leaves the freedom to choose the IDE for that language. If the source code is stored as plain text, it is easy to open it in a different IDE without the need to convert anything.

But if the developer/team chooses a specific structured IDE, the dependency is much greater. Not only the source code will probably be stored in a custom format of this IDE, the developers might get used to the concrete editor and it would be difficult to change the IDE in the future. However, the problem is solvable. An IDE might (and most probably will) provide an option to store source code in text files, just like traditional IDEs do. Or the storage format could be standardized, just like a programming language syntax is standardized. Beside standardizing programming languages of the future, it will probably make sense to simultaneously standardize the structured storage/representation format for programs in that language.

1.5.5. Preserving source code formatting

Many developers would feel very uncomfortable if the custom formatting of their source code would be influenced or even completely lost by an editor. Imagine a situation where existing source code is stored in text files with custom whitespace formatting and comments. Such formatting (as indentation or empty lines) could carry important information for the developer, or even convey vital information via the team's coding conventions.

The problem is that a structured editor is typically unaware of whitespace and formatting because formatting is not formalized by the language grammar and is ignored at the scanning phase.

However, this problem could be solved by using a **formatting preserving parser**, which not only stores text position (line and column) information for every node of the AST, but also stores special nodes that represent all the textual whitespace between the language constructs, which is normally being ignored by traditional scanners.

This formatting information should be taken into account by the structured editor and the pretty-printer should output the source code including all the whitespace.

An interesting research direction could be automatically deducing whitespace formatting rules from the existing source code and auto-inserting such whitespace when new nodes are created in the structured editor. For example, the editor should notice that there is always an empty line between methods of a class, so it should automatically guess and insert an empty line around a newly created method. However this task is obviously very complicated because of its non-deterministic nature and also because whitespace formatting is something that is being mostly ignored by the compiler and programming language research.

1.5.6. Standardizing difficulties

It is already difficult enough to come up with a standard for a programming language, including its grammar and semantics.

Most often standards do not cover code representation (formatting and coding guidelines), because this is not formalized as good as the language syntax.

Coming up with standards for a structured editor for a language is a highly challenging and vastly non-trivial task. Such a standard would need not only to cover the appearance of all language constructs in all possible contexts, including margins, padding, colors, border thickness, styles of backgrounds etc. but it would also have to describe and formally specify the editing behavior, keystrokes, visual changes in display and so on. Such a specification would probably be more voluminous than the HTML 4.0 specification, because HTML only covers static presentation and does not include complex runtime behaviors exhibited by structured editors.

2. Existing research

2.1. History

Decades ago many developers started thinking about alternatives to representing programs as plain text. With the improvement of programming languages and environments, the intuition grew stronger that the syntax tree should be directly represented on the screen. However, the first wave of research and development failed to produce a structured editor or environment capable of becoming mainstream. This failure resulted in a widespread disappointment in structured editing overall, because the promise was very high, but the outcome was unusable. Since the first wave of failures, less effort has been made to build such an environment. But there is still ongoing research happening in this area.

2.1.1. Motivation

Wesner Moise outlines great motivation points for structured editors in his blog posts [WesnerM1] and [WesnerM2] dated 2004.

A good summary of the features of a hypothetical structured editor and a lots of bold ideas are given by Roedy Green in [SCID].

Martin Fowler ([Fowler]) has written an outstanding article on “Language Workbenches”, which is very close to the idea of a structured editor. This article is an in-depth overview about domain-specific languages and the need for the tools to get more language-aware. Sergey Dmitriev of JetBrains writes about “language oriented programming” in [LOP].

2.1.2. Problems with building structured editors

Many (if not all) early attempts to build a usable structured editor failed to become mainstream. One reason for this could be that human-computer interaction wasn't mature enough at the moment (first developments of GUI were not sufficient, many GUI concepts weren't invented yet).

Another complication was the complexity of making a parse tree interactive. This task is fairly similar to building an HTML renderer, but it is more complex in a sense that it needs to support real time editing – just like WYSIWYG editors do. A fairly complex framework of visual controls is necessary. Tying together the functionality of stand-alone controls in a convenient way is not a straightforward task too. A lot of difficult problems need to be solved here: defining an intuitive appearance and behavior of controls, developing fast and robust layout algorithms, defining the order of keyboard navigation, routing mouse and keyboard events, response to user input, modeling transactional undo/redo behavior and much more.

Another important cause why structured editors still aren't mainstream today is given by Wesner Moise in his blog post [WesnerM1]:

I think a major cause of the delay in this revolution is that both C/C++ relied on preprocessors and headers. Some historical languages like Smalltalk actually had this support. Fortunately, more modern languages like C#, Java and VB are standalone files, one class per file, with a little or no preprocessing support. This enables easy parsing.

Most probably this is yet another reason why appearance of managed environments such as Java and .NET actually triggered a new wave in research and development in the area of structured editing.

Another trigger were probably more mature modern developments in the area of the graphical user interface, which provide visualization and interaction techniques that simply weren't available 20 years ago (just think about Microsoft IntelliSense and the impact it made! Or compare Eclipse to Notepad!).

2.1.3. Usability problems

Despite of the complexity to create a structured editor, the first editors were actually built, and they were working, but the usability wasn't good enough to compete with traditional

text editors, which allow for a reasonable editing speed and comfort. Usability is really a major problem, because of two issues. First, almost every editing action requires a sequence of keyboard/mouse operations, often longer than what it takes to edit plain text. Second, this new way of editing is totally different and needs to be adopted by the user first.

However, the research in structured editors isn't dead. A lot of fruitful research is going on currently and promising technologies are emerging, which will hopefully tackle the usability issues. Here we will try to give an overview of major happenings in the area of structured editing.

There are existing implementations of structured editors. We'll consider the most popular of them, in random order.

2.2. Intentional programming

Intentional Programming is the brainchild of Charles Simonyi. Originally started within Microsoft Research and enthusiastically led by Simonyi, the development later branched off Microsoft and became independent IntentSoft Corp. in August 2002 [IntentSoft].

Intentional programming is all about capturing the intentions of a developer during the coding process and maintaining the high abstraction level inside the environment. Thus, the meaning isn't stripped off the code so the environment literally "knows" about what's being edited. Custom editing operations can be defined for the code, which allows the developer to extend the IDE itself.

A great overview of intentional programming is given in the book "Generative Programming" ([CzEi], Chapter 11). Other resources include an insightful interview with Simonyi ([Simonyi1]), an OOPSLA 2006 presentation ([IntentSoft2]) and many others. Links about intentional programming are being accumulated at <http://del.icio.us/KirillOsenkov/IntentionalProgramming>.

An implementation of intentional programming implies a structured editor for the source code. The OOPSLA 2006 paper about intentional programming shows how the model-view-controller architecture is used to interactively display same code in several views – plain text, flow diagrams, etc. However, at the moment of this writing, no public preview of the IP system has been released yet.

2.3. JetBrains MPS

In Sergey Dmitriev’s inspirational article [LOP] a desire is expressed to allow developers to extend an IDE and to be able to customize it for the problem domain. A strong accent is made on developing DSLs – domain-specific languages as opposed to using general tools and frameworks. This approach is called language oriented programming.

The interest of JetBrains in this topic is not only of theoretical nature. They develop a very successful and intelligent IDE for Java, IntelliJ IDEA ([IntelliJ]). It is distinguished by its extensive language-awareness and a large number of full-automated refactoring capabilities. IntelliJ IDEA is also known for its extensible object-oriented architecture.

It is not surprising that having such a good base, JetBrains also work on an implementation of language oriented programming, what they call MPS (meta programming system). MPS is an IDE extension which allows the user to define own domain-specific-languages and to create documents in these languages using a custom-tailored structured editor. These documents (called “models”) can be transformed to generate artifacts such as source code or XML.

Martin Fowler demonstrates the creation of an Agreement DSL with MPS in [Fowler2]. This is a set of a customized editor, language definition and generation rules to generate Java code out of models. This language could be used to generate Java code out of more concise descriptions in a specially tailored language instead of directly encoding it in Java manually. Thus the level of expressiveness is greatly raised.

MPS employs the technique of structured editing to provide an editor for custom languages and language extensions. The editor is universal, meaning it can be configured by a specialized grammar – that is, you don’t need to hard-code the implementation of an editor for each language, but simply edit the language definition (in a special DSL) and the editor will automatically learn how to edit programs in that language. Thus, an editor exists to build editors.

Here we notice how a domain specific language is used to describe grammars for domain specific languages. This idea of a “metamodel”, a language for defining languages, comes up fairly often in the field of meta-programming and generative programming. If we talk in terms of defining specialized languages for everything, why not define a specialized language for creating specialized languages?

2.4. The synthesizer generator

Starting in 1978 a syntax-driven editor called Cornell Program Synthesizer was being developed at Cornell University. The Synthesizer Generator [ReTe1] is the further development of it, which is a tool for generating syntax-driven editors based on the specification of the language grammar. Apart from a structured editor for the Ada language (Ada-ASSURED), the development of synthesizer generator seems to be abandoned.

2.5. Other implementations

2.5.1. ProgramTree

ProgramTree ([ProgrTree]) is a commercial application which replaces curly braces in C++ and Java sources by a tree-view-like outlining. An HTML editor also based on this approach is being developed as well. However it seems like ProgramTree isn't truly a structured editor (is not aware of the semantics), but it simulates structured appearance by matching braces in realtime. Thus, the user still edits plain text and it has to be parsed first.

2.5.2. Lava and LavaPE

As stated at the [Lava] website:

Lava is an experimental object-oriented rapid application development (RAD) language with parameterized ("virtual") types, refactoring, and extensive static checks. The Lava programming environment LavaPE replaces text editors completely by structure editors.

Lava seems to not only present a new programming language and a purely structured editing environment, but also a new programming paradigm – the program should be “composed” instead of being “written”. Lava is an open-source project hosted at SourceForge and is a playground for some interesting ideas and visions of its developers.

2.5.3. BoxView Eclipse plug-in

Eclipse platform has a plug-in developed for it ([BoxView]) that displays a hierarchy of embedded blocks *beside* textual code. This is an interesting approach, which is mostly used for easy selection of language constructs. However, it might be difficult for users to switch

between two parallel views. BoxView currently does not offer displaying the code *in* the blocks.

2.5.4. Other links:

An up-to-date list of web links about structured code editing is maintained at <http://del.icio.us/KirillOsenkov/StructuredEditors>.

2.6. Summary

It is very common for most structured editor implementations that the editor is generated from a grammar using a universal generating tool, as opposed to hard-coding the editor manually. Such systems consist of a universal generic editor generator, which accepts a language specification as input and generates the sources of a structured editor as output.

Instead of writing a structured editor, they write a universal factory for producing structured editors.

Although this approach is one meta-level higher, and thus may occur flexible and universally applicable, in practice it is probably a disadvantage because of tremendous complexity to create such a meta-system.

An intuition based on some experience suggests that hand-written editors could be better customized to be more comfortable, because the creators can manually adjust it to the target language. A generator puts tight constraints on the generated editors, unless there is a powerful extension mechanism which allows for manual extensions.

It also seems more logical to first create an editor manually, adapt and optimize it for better usability, and only after that it makes sense to build a universal tool to generate editors. Both JetBrains MPS and the Synthesizer Generator may suffer or may have suffered from this additional complexity, whereas developing editors manually would probably be a simpler and more manageable task.

3. Functionality of the structured editor

Regardless whether a stand-alone application or integrated into an IDE, the main element of a structured editor is the editor control. An editor control is a user interface element which visually displays the edited program on the screen. It is a rectangular area on the screen where a program or a part of it is displayed.

In case of the structured C# editor presented in this thesis, the editor control can display a single compilation unit at a time.

Theoretically, with a structured editor it is possible to provide different views on the source code. Examples include showing a list of methods in an editor control (e.g. a list of all overrides of a given virtual method), or a single class with no method bodies, or any other arbitrary view on part of the code, all live mapped to the AST. However, only a view of a single compilation unit is supported within the scope of this work.

Here is how an editor control looks like (one of possible visualization modes):

```

using System
    System.Collections.Generic
    System.Text

namespace GuiLabs.Editor.Test
    public static class Program
        public static void Main()
            Console.WriteLine("Hello World")|



```

Figure 8 - the Hello World program in the structured editor

3.1. Creating a program

3.1.1. Hierarchy of blocks

A program is represented in an editor control as a hierarchy of embedded rectangular blocks. There are simple blocks (with no embedded blocks) and there are container blocks, which can contain other blocks.

If a container block has a minus  icon next to it, it can be collapsed by clicking this icon. To expand it again, click the  icon. Most containers, if selected, can also be collapsed or expanded by pressing the [Space] key.

3.1.2. Insertion point

An insertion point is a special location within the editor control where the user can put the caret. The caret indicates that it is possible to insert something at the current caret location. Moving the caret means jumping to the next or previous insertion point. For instance, when editing text, an insertion point is between each two neighbor characters.

3.1.3. Language constructs

In this section we cover the language constructs supported by the current implementation of the editor. We divide the language constructs into two main groups: language constructs that are “**outside**” a method body (such as namespaces, types and type members, as well as methods themselves), and constructs that are “**inside**” a method body (such as statements or expressions). Formally, if we consider the C# grammar, a construct is “inside” a method

if it can be contained within member-body or accessor-body. This separation is important because it divides the language into two approximately equal “levels” – the upper “level” and the “lower” level. We will work with language levels throughout this thesis.

For the language constructs currently supported by the implementation of the editor, we will provide a grammar definition. It will always be a subset of the actual C# 1.0 grammar given in the ECMA-334 standard ([ECMA]).

Language constructs are visually represented by the blocks mentioned above.

3.1.4. Hybrid editor

The structured editor presented in this thesis is not purely structured. It is more like a hybrid between the structured and text editor – higher level of the language (types, members) is implemented in a structural fashion, while lower level (statements, method parameters) is implemented as plain text.

It seemed to lower the usability if all language constructs, including statements, were implemented differently from text.

3.1.5. Compilation unit

Here is the definition of the compilation unit as supported by the current implementation of the editor.

```
compilation-unit:
    using-directivesopt namespace-member-declarationsopt
```

At any given time the editor control can contain a single compilation unit. A compilation unit corresponds to the `compilation-unit` non-terminal of the C# grammar and is visually represented by a vertical list of blocks. These blocks could be comprised of using directives and namespace member declarations. There can be only one using declaration, at the top of the compilation unit.

3.1.6. Empty blocks

The elements in this vertical list are blocks. Between each of these blocks, there is a so-called **empty block**, which separates the language constructs from each other with some whitespace and allows to insert new language constructs in place of the empty block. Each

empty block is an insertion point. Empty blocks explicitly model a non-terminal of a context-free grammar, which allows for insertion of language constructs in the correct order. Let's illustrate this with a context-free grammar with two non-terminals:

```
A → dB | AcB
B → BcB
```

"A" here is a non-terminal (empty block), from which both using directive "d" as well as a namespace member declaration "c" can be produced. We see, that as soon as we have produced a using block "d", it appears at the beginning and no further using blocks can be produced. If we choose to produce a namespace member declaration out of "A", we still have the ability to insert a using block before it, but not after. "B" represents an empty block, from which only namespace member declarations can be produced. One cannot produce a using block from such an empty block anymore.

This mechanism of empty blocks automatically ensures that only grammatically correct compilation units are allowed. One cannot create a using block after a namespace member declaration, neither can one create a second using block after the first one has been created. Additionally, one cannot create a namespace member declaration before an existing using block, but can freely create it before or after any other namespace member declaration.

3.1.7. Inserting new language constructs with empty blocks

The user of the editor can position the caret inside the empty block to insert a new language construct at this position. After insertion, it will be automatically surrounded by appropriate empty blocks.

Each empty block has a list of all language constructs, that can be inserted at its position. To show this list, the user can start typing – the list will popup and highlight the desired alternative:

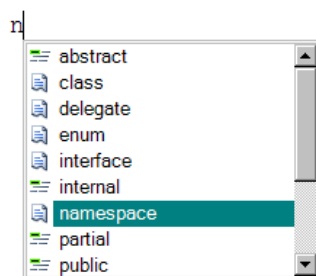



Figure 9 - completion list inside an empty block

Otherwise, the user can press [Tab], or [ContextMenu] or right-click with the mouse at the caret location (to the right of the caret). To select an alternative from the list, the user can press [Space], [Enter], [Tab] or left-click the desired item with the mouse. At the moment when an alternative is selected, new blocks are created according to the grammar rules.

Here is an example how a new namespace can be added to the empty compilation unit. An empty compilation unit always contains a single empty block, which serves as the starting point for the editing. The caret is positioned to this empty block. The user starts typing “na” and the completion list appears with the word “namespace” highlighted (Figure 9).

The user presses [Space] to insert a new namespace:



The caret jumps automatically to a place where the user can input the namespace name. The user types in a name for the namespace and can press [Return] or [DownArrow] to move to the namespace contents. One can also enter the name at a later stage and navigate away right after creating the namespace – this is allowed. One can always come back later and complete the definition by entering the name, or change the name at any time.

3.2. Namespace member declarations

A namespace member could be a namespace or a type declaration.

```
namespace-member-declarations:
    namespace-member-declaration
    namespace-member-declarations namespace-member-declaration

namespace-member-declaration:
    namespace-declaration
    type-declaration

type-declaration:
    class-declaration
    struct-declaration
    interface-declaration
    enum-declaration
    delegate-declaration
```

3.2.1. Namespace

As we have seen above, a namespace is represented by a container with the “namespace” keyword, a textbox for a namespace name and a vertical list of “children”.

3.2.2. Navigating containers with the caret

Navigating the caret around the namespace is the same as for all other containers. Arrow keys move the caret or select the entire namespace block.

When the caret is above the container block, pressing the [DownArrow] key first selects the block itself. When a block is selected, pressing the [RightArrow] key selects the first element in the title row of the container (the name). Pressing [DownArrow] key selects the first child of the container. Generally, [UpArrow] and [DownArrow] traverse the block tree in the depth-first search order. See more about this in section 6.2.

[LeftArrow] key is opposite to the [RightArrow] key, while [UpArrow] key is opposite to the [DownArrow] key. In general, navigating the block tree is very similar to navigating caret in plain text, with two major exceptions:

1. The caret in a structured editor can only be placed where it makes sense (where there is an insertion point available).
2. An additional stop in the caret movement is made to select the whole container when a caret is moving into it. Each container precedes all its children in the traversal order.

3.2.3. Using directives

To add a using declaration, the user positions the caret to the empty block above the namespace (by pressing the [UpArrow] key) and starts typing “u” – the “using” item is highlighted in the completion list. [Space] inserts a new using declaration. The using declaration consists of a single container per compilation unit, with the word “using” on it. The container contains a vertical list of strings – namespace names, each namespace name on a separate line. A semicolon at the end of the lines is not required, because the editor doesn’t (currently) allow multiple declarations on the same line.

```
using |  
  
namespace Test  
|
```

Figure 10 - adding a using declaration

The user can now type in a namespace name to import. Using directives are placed inside a single 'using' container (only one container per compilation unit). When done entering the namespace name, pressing [Enter] moves the caret to the second line inside the same 'using' container. When no further using directives are desired, pressing [Enter] for the second time, [Backspace] or [Shift+Tab] exits from the using container and moves the caret to the next empty block:

```
using System  
| System.Collections  
|  
namespace Test  
|
```

Figure 11 - exiting from the using declaration

3.3. Type declarations

The user of the editor can add each of the 5 supported type declarations to the program: a class, a struct, an interface, an enum or a delegate. There are two ways to add a new type definition. The first is explicitly choosing one of “class”, “struct”, “interface”, “enum” or “delegate” from the completion list of an empty block:

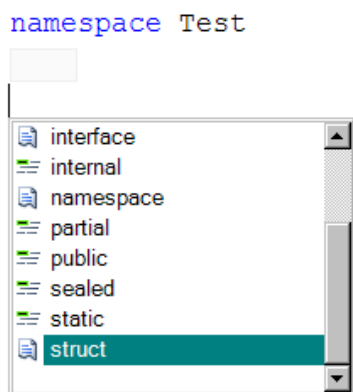


Figure 12 - completion list for creating types

The second is choosing an access modifier first. After an access modifier has been entered, the editor transitions into a temporary state which allows entering further access modifiers or a type declaration. This way, the user can enter text just like in traditional text editors: “pu” ([Space] key completes to “public”) “st” ([Space] key completes to “static”) “c” ([Space] key inserts a new public static class).

Blocks that represent definitions of class, struct, interface and enum are all similar containers, with a horizontal title line and a child compartment:

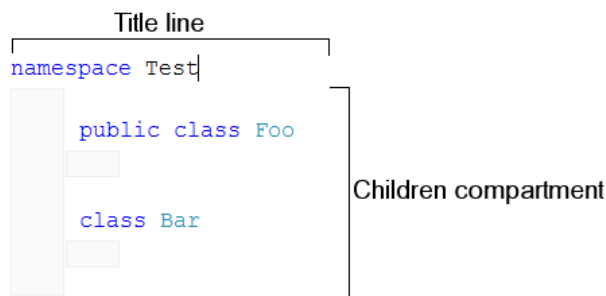


Figure 13 - container block sample

Unlike namespace blocks, type definition blocks have an additional section that allows editing the access modifiers of the corresponding language construct.

3.4. Access modifiers

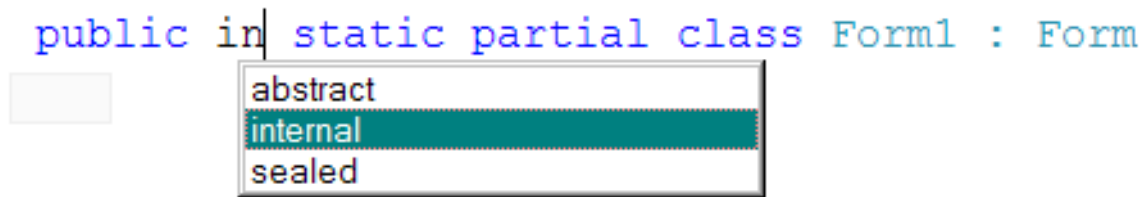
A special element of the user interface is used to model access modifiers. It is a horizontal list of access modifier keywords and spaces between them:

```
public static partial class Form1 : Form
```

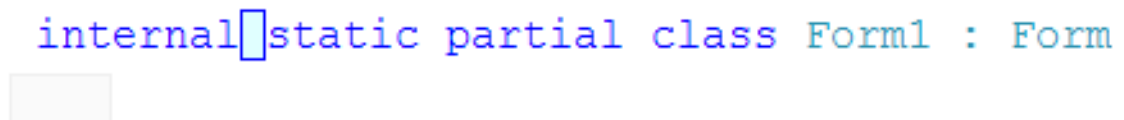
Although the modifiers may look like plain text, they are not. A space between two access modifiers can be selected and is actually an insertion point:

```
public static partial class Form1 : Form
```

The user can right-click the selected insertion point to popup the completion list or just start typing in a modifier:

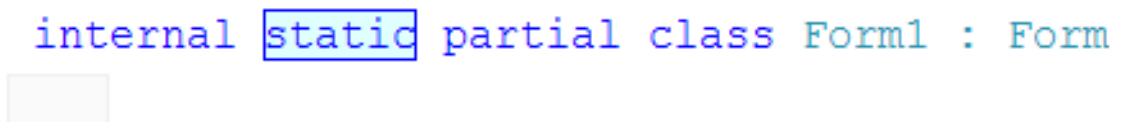


As we can see, only the modifiers which are valid for a class are being displayed in the completion list. Once the user commits the completion list by pressing [Space], [Enter], [Tab] or by clicking the desired completion list item, the access modifiers of the class are changed accordingly:

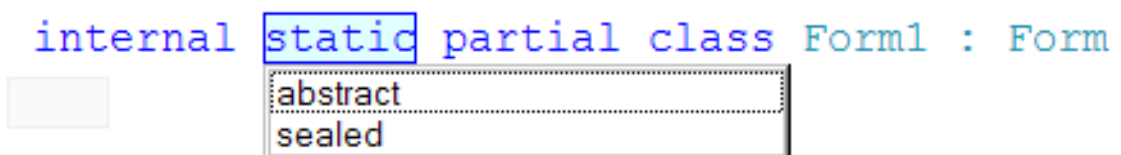


The editor didn't simply insert the word "internal", it actually changed the modifier at the underlying data structure, and the view was updated accordingly to reflect the fact that the class is not public anymore, but internal. It doesn't matter, which insertion point we choose to insert a new modifier – it could be before or after any existing modifier keyword. If a modifier already exists, it will not be added a second time. If a modifier just entered overrides some existing modifier (just like internal implies not public), the old conflicting modifier is deleted automatically. New modifiers are automatically inserted at the correct location, regardless where the caret currently is.

Modifier keywords can only be selected as a whole:



When a modifier keyword is selected, the user can just start typing or press [Enter] to popup a completion window with a list of possible replacements:



To delete a modifier keyword, select it and press [Delete]. If an insertion point between two modifiers is selected, pressing [Delete] deletes the next modifier keyword, if available, and pressing [BackSpace] deletes the previous one.

All changes to the modifiers have a transactional nature and can be undone if necessary.

3.5. Class and struct members

The possible contents for classes and structs supported by the current implementation are the following:

```
class-member-declaration:
    field-declaration
    method-declaration
    property-declaration
    constructor-declaration
    static-constructor-declaration
    type-declaration

struct-member-declaration:
    field-declaration
    method-declaration
    property-declaration
    constructor-declaration
    static-constructor-declaration
    type-declaration
```

3.5.1. Method

A method is represented by a container with the title line and the children compartment. The title line is a horizontal list that contains access modifiers, method return type, method name and parameters. One can create a new method by choosing the item “method” from the completion list of an empty block. One can only create methods in empty blocks that reside within a class or a struct.

Just like with type definitions, the second way to create a method is just starting to type access modifiers or the method return type. After an access modifier or a return type has been entered, the editor is in a special state which allows entering the member name. Only after the user has finished entering a member name and presses “(“, the editor can decide that the user wishes to enter a method. If the user presses “{“ instead, a property will be

created. Until these two decisions are made, the unfinished string is considered a field declaration (unless the type is “void”).

The children compartment of a new method initially contains an empty statement block. An empty statement block is a textbox where the user can enter new statements. When a syntactically correct statement is entered into an empty statement block, it becomes a statement block itself. As soon as the text of the block is edited and doesn’t represent a syntactically correct statement, it is again treated as an empty block.

3.5.2. Property

Properties have always been a very useful and an often used feature in C#. However the syntax to enter a property in C# 1.0 is a little verbose: it is necessary to enter 12 lines of code for the trivial property:

```
private string mName;  
public string Name  
{  
    get  
    {  
        return mName;  
    }  
    set  
    {  
        mName = value;  
    }  
}
```

Visual C# 2.0 provides aid to simplify entering such properties: there is a “prop” Code Snippet, which simplifies entering the property to a minimum effort. However, once the property has been entered, working with it still involves a lot of typing.

Finally, in C# 3.0, a new feature appears, which allows automatic generation of trivial property implementation by the compiler. Thus, it is sufficient to enter:

```
public string Name { get; set; }
```

to achieve the same purpose as the code above.

Working with properties in the editor presented in this thesis is simplified. While there is a “prop” item in the completion list, which results in an empty property being inserted, a more flexible way to create a property is to start typing just like if creating a method, and then pressing the “{” key after the property name or on the next line:

```
class Program
    public string Name
```

becomes

```
class Program
    public string Name
        get
        set
```

where the `get` and `set` accessors are inserted automatically. To delete a `get` or a `set` accessor, it is sufficient to select it and press the `[Delete]` key, just like with any other block. If both `get` and `set` are deleted, the property becomes a field declaration. When the user presses “{” in the next line after a field declaration, the field becomes a property with both `get` and `set` accessors. If only one of the accessors is present, and the user wants to add the second accessor, it is sufficient to select the existing accessor and to press `[Enter]` or `[Insert]`.

3.5.3. Field

Another type of a class/struct member is a field. As stated above, creating a field is similar to creating a method or a property until the user presses “(“ or “{“ keys to turn the incomplete declaration into a method or a property, respectively. Thus, adding a new field involves typing in the keywords for the access modifiers, the type of the field and the field name. One can also enter a field initializer after pressing “=”. When a field initializer is present, one cannot convert the field to a property or a method anymore.

Unlike properties or methods, a field block is not a container, but a single line of text and keywords. The field can be selected by pressing `[Home]` or moving the cursor to the left of the first element of the field. Alternatively, left-clicking with the mouse in a blank area to the right of the last field element selects the entire field as well:

```
public int x = 5
```

Figure 14 - selecting a field

Just as for any other block, selecting the field is useful, for example, to delete it with the [Delete] key.

3.5.4. Constructor

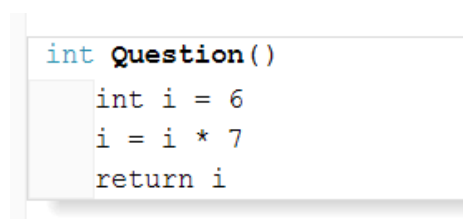
The constructor can be inserted in a class or a struct by choosing the “ctor” item from the completion list. A constructor differs from a method in a sense that the name of the constructor is already predefined by the editor and is the same as the name of the nearest containing class. Moreover, different constraints are specified for the access modifiers (a constructor cannot be virtual or sealed). However, a constructor still can be static. A static constructor cannot have any further access modifiers. As soon as the constructor with an access modifier is made static, all other access modifiers are deleted automatically. The editor automatically ensures that only available access modifiers are shown in the completion list and can be inserted.

3.6. Statements

The editor differentiates between two types of statements – those that can contain embedded block statements (e.g. control structures) and those that are normally written in a single line of code in text editors (e.g. assignment, variable declaration, return statement, etc). Assignment, variable declaration and the return statement are represented like usual lines of text. It was not possible to find a reasonable non-text representation for such statements within the scope of this thesis. Besides, the text representation for such items is good enough and the author didn't see any possibilities for improvements.

The editor represented in this thesis can thus be classified as a hybrid editor, which models most language constructs as blocks, but still represents certain concepts as plain text.

Here is a code example that demonstrates using text-based statements inside a method block:



```
int Question()  
    int i = 6  
    i = i * 7  
    return i
```

Figure 15 - statements as text

Editing such statements is based on the same principles as editing a plain text program. Pressing [Enter] at the beginning of such a line inserts a new empty line before the current one. Pressing [Enter] at the end of the line inserts a new empty line after the current line.

3.7. Control structures

One can embed control structure blocks, such as a for-loop, directly between the statement lines. Here is an example of a for-loop between the text statement lines:

```
int Question()  
    int i = 0, k = 0  
    i = i * 7  
    for j = 0; j < i; j++  
        k = i * j  
        k++  
    return i
```

Figure 16 - example of a for loop

At the beginning of each text line within a method body, a completion list item is available for each control structure: for-loop, foreach-loop, while-loop, as well as if and else statements. Just like namespaces, classes and methods, the control structures are represented by container blocks with a title line and the children compartment. Each children compartment is initially an empty statement block itself. A title line usually contains the keyword that describes the control structure as well as additional text information if necessary (for example, a boolean expression for an if-statement).

3.7.1. for statement

The title line of a for block contains three textboxes separated by two semicolons:

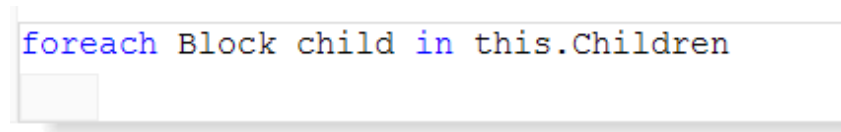
```
for | ; ;  
    
```

One can move the cursor between the textboxes just like if it was plain text – with the left and right arrow keys. Pressing the [Tab] key moves the cursor to the next textbox.

The semicolons are not editable characters – one should treat them as passive separators that aren't affected by the editing process. When the caret is at the end of the first textbox and the user presses [Space] or [RightArrow], the caret moves to the beginning of the second textbox. The same for the end of the second textbox – it is connected to the beginning of the third textbox.

3.7.2. foreach statement

The foreach block is similar in structure to the for block:



```
foreach Block child in this.Children
```

Figure 17 - foreach block

The title line of the foreach-container has three textboxes: for the type of the enumeration variable, for the name of the enumeration variable and for the object that implements IEnumerable (typically a collection or a list). When the caret is at the end of the first textbox, pressing [Space] or [Right] moves the caret to the beginning of the second textbox. When the caret is at the end of the second textbox, pressing [Space] moves the caret to the beginning of the third textbox. When the caret is inside any one of the textbox, pressing the [Tab] key moves the caret to the next textbox (cyclic, i.e. from the third textbox the caret moves to the first again). When the caret is in any of the textboxes, pressing [DownArrow] or [Return] moves the caret to the children compartment. It is impossible to delete the “in” keyword, it is hard-coded into the title line of the foreach block, just like semicolons are built into the title line of the for-block.

3.7.3. while, if and else containers

while, if and else control structures are all implemented very similar to for and foreach, but their title line is simpler and only contains a single textbox for an expression.

3.8. Comments

Although there are great opportunities to implement convenient, rich and active comments in a structured source code editor, the general complexity of this work as well as time and

resource constraints didn't allow to implement comments in the current release. Comments were prioritized out to allow implementing the main functionality. For some inspiration on how comments could be implemented, please see for example [LutzR1].

Advanced commenting capabilities could make many documentation and commenting methods obsolete, because texts, hyperlinks and diagrams could be inserted directly into the source code document. XML Comments could boast fill-in-the-blanks approach.

3.8.1. Types of comments

Traditional editors use the notion of a comment for different purposes:

1. Explaining a line (or lines) of code
2. Temporary disabling ("commenting out") portions of code
3. Documenting classes, methods, etc. with XML comments

Given a structured editor, it probably makes sense to explicitly implement different functionality for each purpose mentioned above. Explanatory comments could be implemented as floating or carry-out clouds that could be switched on or off. Every connected code segment could be enabled or disabled using special controls.

XML comments could be embedded into the main document flow or carried out to a separate properties window (XML document properties window). When a block is selected in the main editor, its XML comments (and possibly other meta-information) could be shown in a separate window.

3.9. Code completion

A structured code editor implemented in this work is actually a component which can be used both in a stand-alone application or integrated into an existing IDE. In both cases, the environment into which the editor control is embedded, could provide context-sensitive information about the program being edited. Microsoft calls this technology "IntelliSense", and the engine that provides IntelliSense is called a "language service".

Once a language service is available for the editor control, it can provide two main code completion features: showing a list of available members of a type after the user pressed the "." key, or showing the information about parameters of a method after the user presses the "(" key.

4. Architecture

4.1. The Editor Framework

The nature of a structured editor suggests that the user interface is composed of different details – interactive blocks, that together form a document. Each block (a control) represents a language construct and can visually contain other blocks. When implementing these blocks, it is wise to carry them out into a separate library, so that the same basic building blocks can be re-used for different editors.

Indeed, such a library of blocks has been created as a foundation for creating structured editors¹. From an architectural point of view, this library can be called a framework, because it provides APIs and reusable code to easily define custom blocks. The framework allows to model syntactic structures and to provide an interactive visual representation of such structures.

A structured editor is a .NET component (a user control) which depends on this framework. It could be integrated into an existing IDE or hosted in a stand-alone .exe file. The framework supports the development of structured editors using the Microsoft .NET Framework (version 2.0 as of June 2007).

¹ The editor framework was designed and developed by Kirill Osenkov, Steffen Büchner, Alexander Kapitanovskiy and Stefan Adam as part of the research effort at the Chair of Programming Languages and Compiler Construction supervised by Prof. Dr. rer. nat. habil. Peter Bachmann

4.1.1. Graphical Controls Library

An important part of the editor framework is a generic graphical library which provides capabilities to visualize shapes, embed, layout and auto-resize shapes contained in other shapes, process mouse and keyboard input, etc. Such a library must satisfy certain requirements in order to be suitable for creating a structured code editor. The most important requirement is being able to easily manage dynamic content – automatic placement and layout of newly created shapes, automatic repositioning of neighbor shapes during resize, etc. While choosing such a graphical library, a decision was made not to employ Windows Forms, because Windows Forms provides too heavy-weight and resource-intensive (OS native) controls which are not well suited for dynamic positioning and auto-layout of dynamic documents. While Windows Forms may be good for static content like dialogs and forms, it would definitely prove problematic for such highly dynamic and interactive content as structured source code. Moreover, the experience of extending Windows Forms controls has shown, that because they are just wrappers around native Windows UI elements, the extensibility of these controls is limited.

Just as Microsoft Word doesn't use Windows controls for WYSIWYG editing and just as web-browsers custom-implement the entire rendering functionality, all structured code editors (based on this editor framework) use a custom drawing library provided by the framework, which was specially designed with dynamic content in mind.

During the planning phase (2004-2005) of the entire project a decision was made to implement a graphical controls library from scratch. If at that time the release of Windows Presentation Foundation [WPF] was available, most probably it would have been chosen instead, because it provides an excellent object-oriented library of visual controls well suited for dynamic content. Unfortunately, WPF wasn't available at that time.

4.1.2. Summary of users and roles

We distinguish three major roles:

1. **Developers of the editor framework** provide several .dll files (.NET assemblies), where all the core functionality and base blocks are defined. The author of this thesis belongs to the group of the framework developers.
2. **Users of the framework** develop a structured editor (*target editor*) for some programming or markup language (*target language*). The framework .dlls are used

as a reference. Hence, the users of the framework are most often *authors* of an editor.

3. **Users of the target editor** work with the editor to create and edit programs or documents in the target language. They don't necessarily require any .NET IDE or .NET knowledge for this purpose (unless the target editor is for one of the .NET languages).

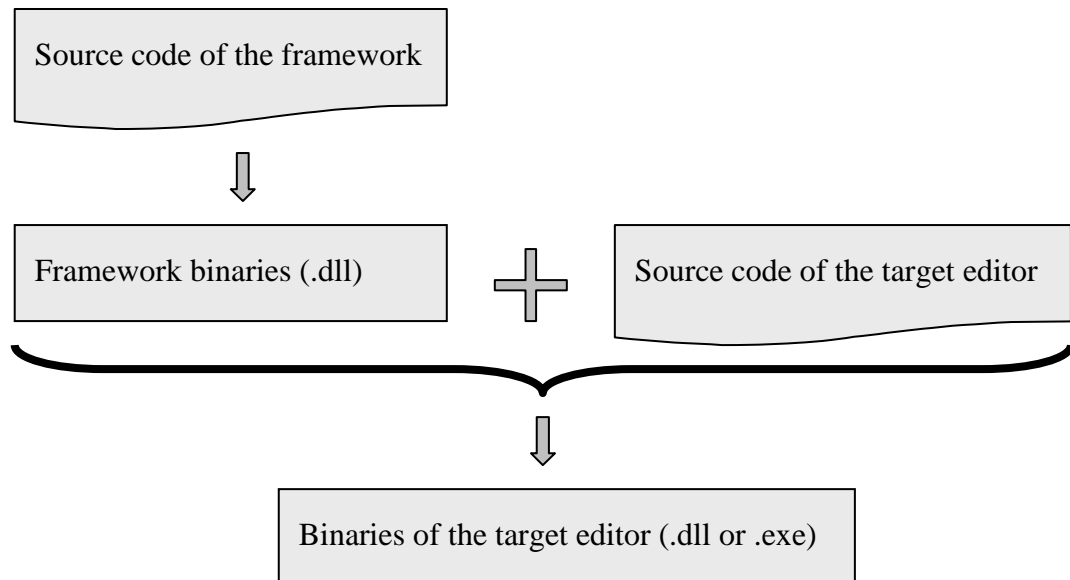


Figure 18 – Dependencies of an editor from the framework

When we speak of the structured editor framework, we can mean both the framework sources as well as the compiled binaries, depending on the context.

The framework design is based on a layered architecture and consists of four layers. Each higher layer depends on all the layers below it. Each layer is represented by a C# 2.0 Project:

4.2. Utils

The `Utils` project has no dependencies on other layers. It defines different helpers and auxiliary code used throughout the framework and target editors. `Utils` provides classes to simplify work with the file system, calling Win32 API, colors, strings, lists and collections, timers, common delegates. It also provides base classes and interfaces to model actions and user interface commands.

4.3. Canvas

`Canvas` is a low-level presentation layer (graphical library), which provides drawing capabilities. It only depends on the `Utils` layer.

An important design decision was not to explicitly use GDI+ or any other graphical library for drawing purposes directly, but instead to create an intermediate abstraction layer between the framework and different graphical backends. `Canvas` currently provides rendering options using GDI and GDI+. DirectX and OpenGL renderers could be implemented using the same principles.

As measurements have shown, GDI renderer provides better performance while rendering simple geometric 2D content. That is why the GDI renderer is the default renderer of the `Canvas` library.

An important class in the `Canvas` library is the `DrawWindow` class. `DrawWindow` is a user control that provides the `Repaint` event. When the users want to draw something on the surface of the `DrawWindow`, they subscribe to the `Repaint` event and draw everything there. The `Repaint` event has one parameter of type `IRenderer` – subscribers of the event use this parameter to access drawing functionality.

`DrawWindow` is the Facade class to the `Canvas` library.

4.4. Controls

As stated above, another design decision was not to use Windows Forms to model blocks visually. The `Controls` library defines custom classes that model visual shapes on the screen (these shapes are also called *controls*). These controls have no dependence on `System.Windows.Forms` and are rendered using the `Canvas` library of the framework. They are specifically suited for displaying interactive hierarchical data structures on the screen and allowing the manipulation of the data by user (using mouse and keyboard).

In this thesis, controls mean the objects from the `Control` library of the framework, and not the Windows Forms controls.

The root class of the controls hierarchy is the class `Control`. Commonly used controls like `TextBox`, `Label`, `Button` and `ContainerControl` inherit from the class `Control`.

4.5. Core

`Core` is the main library of the editor framework. It defines the data structure `Block`, which is used to model an arbitrary language construct. One can implement custom language constructs by inheriting from the base class `Block` or some other base class and adding required functionality.

The advantage of the structured editor framework presented in this thesis is that all language constructs can be treated uniformly. Thus the framework can provide services for all (even future) language constructs:

- Hierarchical data structure (Previous, Next, Parent, Children)
- Atomic operations on the data structure (Add, Move, Insert, Replace, Delete)
- Virtually unlimited Undo/Redo steps and a transaction system
- Rendering and scrolling
- Focus and navigation
- Hosting (displaying) the data structure in a Windows Forms Control
- Popup menus and drop-down selection (completion)

4.5.1. Blocks and controls

Each block is represented on the screen using a control from the `Controls` library. Each object of type `Block` has a run-time reference to a corresponding object of type `Control`.

Note about the OO design of blocks and controls

It is important to understand that “a block HAS a control”, and not “a block IS a control”. Controls have no idea of blocks and have no dependency on the `Core` library. This gives us the flexibility and freedom to reuse controls in other graphical projects.

The design decision to implement controls outside of the blocks hierarchy was a difficult one. In early implementations, the class `Block` inherited from the

class `Control`. This turned out to cause problems with `ContainerBlock`, which (conceptually) should be inherited both from `Block` and from `ContainerControl`. Since .NET doesn't have multiple inheritance (and C# has no mix-ins), one has to choose, whether `ContainerBlock` should be inherited from `Block` or from `ContainerControl`. If `ContainerBlock` is inherited from `Block`, the functionality of `ContainerControl` has to be doubled in `ContainerBlock`. If `ContainerBlock` is inherited from `ContainerControl`, the entire code from `Block` must be duplicated.

This problem is known as „burning the base class“. A good example of burning the base class is the `System.MarshalByRefObject` class in the .NET framework. It clearly should have been an attribute instead. Brad Abrams and Krzysztof Cwalina talk about this design problem in [FDG].

The Bridge design pattern helped to resolve the problem by replacing inheritance with aggregation. This decision also provides more flexibility and allows automatic data-binding of a container control to custom lists (see [WPF] for the explanation of data-binding).

4.5.2. Actions

Any change to the block data structure at runtime is separated into two steps:

1. prepare the change (record it into a step description, a so called *Action*) and
2. actually apply the change using this description.

Thus, changes are not carried out directly, but are first encapsulated in Actions (delayed, „lazy“ execution). This architecture allows to save actions in an Undo/Redo buffer (design pattern Command), to keep history and to let users undo or redo any change to the edited document.

4.6. Implementation of the C# editor

This thesis presents a structured source code editor for a subset of the C# 1.0 programming language (as defined by the ECMA-334 standard in [ECMA]). Unfortunately, it was not possible to fully implement even the C# 1.0 version of the language, because this task requires an effort of much more than six man-months. However, the implemented subset is rich enough to provide a good proof-of-concept and to give an example of a working

structured code editor. The editor is based on the structured editor framework described above.

A structured editor is most useful when implemented not as a stand-alone application, but inside a full-fledged IDE. Only when integrated with other features like class browser, project explorer, etc. it can provide the convenience expected from a modern programmer's tool.

The editor presented in this thesis is integrated into SharpDevelop ([SD]), an open-source IDE for the .NET Framework. It allows to parse existing source code into its own internal representation, to work with it and to generate source text again out of it.

5. Blocks

5.1. Data structure

This chapter describes the classes of the editor framework which inherit from the class `Block`. The authors of a structured editor typically inherit from these classes to model their language constructs.

5.1.1. Tree

At runtime of the target editor, the instances of block classes are composed together into a dynamical hierarchical data structure, which is the in-memory representation of the edited program. For instance in a context of a C# code editor, each block tree corresponds to a compilation unit.

In different applications there are many different names for such a data structure: parse trees, DOM, AST, etc.

In a good architectural approach the data structure would have the following properties:

1. The data structure would serve as a model in a Model-View-Controller pattern. As a model, it would be unaware of its (perhaps many) representations (views).
2. The data structure would provide a set of events to notify observers of its changes. This means, every change of the data structure should be tracked by attached listeners (observers) to the nodes or to the whole tree. Such event mechanisms

could provide data-binding capabilities – automatic update of all views by every change of the model as well as keeping everything synchronized.

3. The view would present a tree which is parallel to the model – each node of the view (a control) would map to the corresponding node inside a model.

However, within the scope of this thesis, no such data structure could be created. Designing a generic, observable, abstract syntax tree is a very complex task. To achieve the goal of creating a working proof-of-concept editor, this task was simplified in the following way. The block data structure was given only one view – the corresponding hierarchy of controls. Blocks explicitly know about controls, which is a direct contradiction to the classical MVC pattern. Nevertheless the scheme proved viable enough to create a working code editor. Unfortunately, the scenario of attaching another view to the hierarchy of blocks could not be supported at this time.

There is hope that the manufacturers of IDEs will eventually come up with the MVC architecture for the source code representation, and that the code model will be explicitly separated from the views in the editor windows. Given such a data structure and provided it is observable, one could easily adapt the blocks presented in this thesis to become a view on such a data structure.

5.1.2. Class vs. interface

An interesting experience about designing the block datatype is connected with the choice class vs. interface. At the early stages of framework design all blocks also implemented the `IBlock` interface. When a new member was added to the `Block` class, it had to automatically to be added to the `IBlock` interface as well, because clients were working with the `IBlock` interface. Moreover, there were interfaces for `IContainerBlock` and `IRootBlock`. This turned out to be impractical, because `IBlock` in reality duplicated the same entity which was already modeled by the `Block` class. In [FDG] the authors advise to prefer classes over interfaces when the whole entity has to be modeled. Interfaces are useful when an existing entity has additionally to comply with some contract. Thus, interfaces are good when adding declared functionality to existing entities.

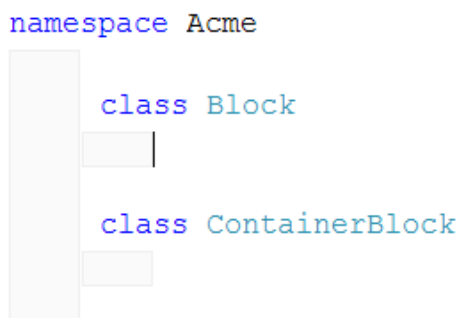
As a result of a major refactoring, all interfaces that duplicate entities were removed from the framework code. This turned out to be a good decision.

5.2. Tree organization

At each hierarchy level, blocks form a doubly linked list. For this, the class `Block` has `Prev` and `Next` properties. The class `Block` also has a property `Parent` of type `ContainerBlock`, which is a reference to the containing parent block.

The `Root` property of each block references the root block of the entire tree. The root of the tree must be of type `RootBlock` or inherited from it. If a block or a subtree of blocks isn't added to any tree (`Root == null`), we say it is “hanging in the air”, or “dangling”. Once a block or a subtree of blocks is added to some tree, the `Root` property of each added block is set to the `Root` of the entire tree.

The class `ContainerBlock` represents all blocks, which can have children, i.e. embedded („contained“) blocks. `ContainerBlock.Children` is a property of type `IChildrenList`. Child blocks are normally visualized as smaller rectangles inside a bigger “parent” rectangle. Here is an example of a namespace block containing two class blocks:



For each block, other blocks in the same linked list are called *siblings*. The blocks in the `Children` list are called *children*. `Parent` and `Root` blocks are called respectively.

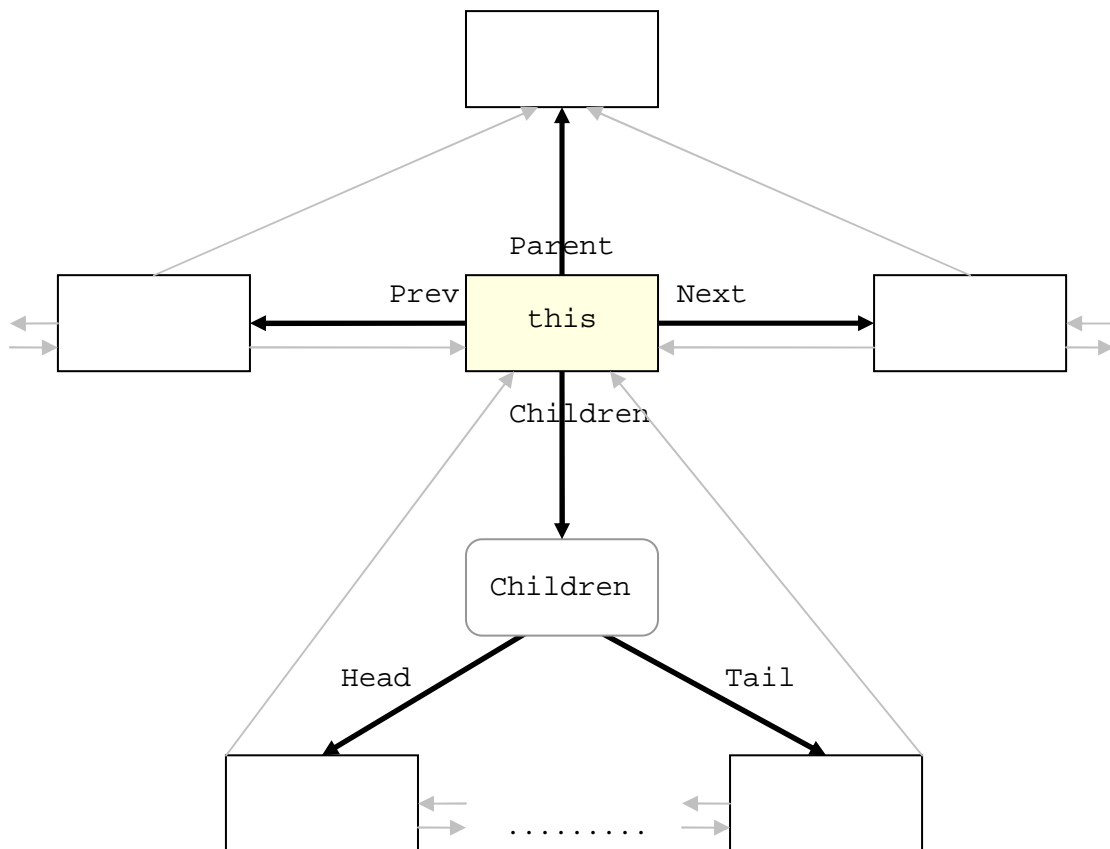


Figure 19 - a block with its siblings, parent and children

The list data structure has proven to be comfortable and efficient. Every modification takes only constant time $O(1)$. Iterating through all of the children takes $O(n)$ time, but in practice (since lists in most situations have less than 10 – 20 elements) this overhead is insignificant. Per-index access is simulated with $O(n)$ time but the situation where per-index access to a child block is necessary is extremely rare.

Taking arrays for Children list would be more memory intensive and less efficient, because changing the data structure and iterating in order happens statistically much more often than per-index lookup, that's why the $O(1)$ lookup time in an array wouldn't justify itself against reallocation costs.

5.3. Operations on the data structure

5.3.1. Inserting blocks

The method `AppendBlocks` of class `Block` appends one or more existing blocks after the current block:

```
this.AppendBlocks(new LabelBlock("Newly added"));
```

This method is overloaded and can accept either an `IEnumerable<Block>` or a parameters array (`params Block[]`). The block or blocks being added should have their `Root` property equal to `null`, otherwise an exception is thrown.

Adding a block which is already added somewhere would actually mean “move”. There is a special operation for moving blocks and it should be used instead.

If the block on which the method `AppendBlocks` was called is contained in some tree (`Root` property is not `null`), an `AddBlocksAction` is prepared and executed¹. Otherwise, the blocks are added directly, because a stand-alone block outside any tree doesn't have access to any `ActionManager`.

The `AppendBlocks` method is marked `virtual` and so can be overridden in subclasses, if necessary.

The method `PrependBlocks` is similar to `AppendBlocks` with the difference that the blocks are inserted before the current block and not after it.

5.3.2. Deleting blocks

The virtual `Delete()` method of a block removes the current block from the list of its siblings.

The block object continues to exist as long as it is being referenced. Later the block can be re-inserted in any other place in any block hierarchy.

¹ See section 6.4 for more information about recording and executing actions.

After being deleted from the tree, the block's `Prev`, `Next`, `Parent` and `Root` properties are all set to `null`. The block starts to “hang in the air”, which means that it doesn't belong to any block tree anymore. When a dangling block is re-inserted into some tree, its `Prev`, `Next`, `Parent` and `Root` properties are updated accordingly. When a `ContainerBlock` is deleted, the `Root` property of every block in the subtree is set to `null`, their other properties do not change.

The call to `Delete()` prepares and runs a `DeleteBlocksAction`.

Sometimes it is required to delete the next block as well. By overriding the virtual `GetBlocksToDelete()` method users can determine the blocks which should be deleted with the current block when it is deleted. The default implementation of the `GetBlocksToDelete()` method is as follows:

```
public virtual IEnumerable<Block> GetBlocksToDelete()
{
    yield return this;
}
```

A good example of overriding this method is in a block which models a namespace in the structured editor. When a namespace is deleted, the next following empty block must be deleted as well, otherwise there will be two empty blocks in a row remaining after the current block. One can override `GetBlocksToDelete()` like this:

```
public override IEnumerable<Block> GetBlocksToDelete()
{
    yield return this;
    if (this.Next != null)
    {
        yield return this.Next;
    }
}
```

5.4. ContainerBlock

5.4.1. Children

`ContainerBlock` is the base class for all blocks which can contain other blocks at runtime. Each `ContainerBlock` has the `Children` property of type `ICollection`, which represents a doubly linked list of all blocks contained in the `ContainerBlock`.

5.4.2. Adding blocks to a ContainerBlock

There are two method groups which allow adding blocks to the container.

1. `Add(IEnumerable<Block>)` and `Add(params Block[])` add one or more blocks to the end of the Children list.
2. `AddToBeginning(IEnumerable<Block>)` and `AddToBeginning(params Block[])` add one or more blocks to the beginning of the Children list.

Overloads with the `params` keyword are provided for convenience, when the user needs to add a single block or a quantity of blocks known at compile time. They are just a shortcut to the other overload which accepts `IEnumerable<Block>`.

All four methods use delayed execution when an `ActionManager` is available, and add blocks directly if the `ContainerBlock` is “hanging in the air”.

One can also operate on the Children collection directly: it provides methods `Add`, `Append`, `Prepend` etc. However these operations are more low-level and local to the list. The changes will take place immediately and the Undo/Redo buffer will not be notified. Such a change will also not be visible to the user until the next screen refresh. The focused (active) block doesn’t change as well.

5.4.3. FindChildren und FindChildrenRecursive

`ContainerBlock` also provides several methods to search for child blocks:

- `FindChildren<T>()` returns all child blocks which can be casted to type `T`.
- `FindChildrenRecursive<T>()` also searches the whole subtree recursively.
- `FindChildrenRecursive<T>(List<T> results)` adds found blocks into the results list, instead of returning them.

This sample code shows how to append an asterisk to the text of every `TextBox` in the current container:

```
foreach (TextBoxBlock textBlock in this.FindChildren<TextBoxBlock>())
{
    textBlock.Text += "***";
}
```

5.5. RootBlock

The class `RootBlock` is a subclass of `ContainerBlock`, which is used as a container for the entire document. Each instance of the class `RootBlock` is at runtime the root of some block tree. Since each `RootBlock` is also a `ContainerBlock` it can (and should) contain child blocks.

The `RootBlock` is an integration point between the blocks data structure and the window, where the block tree is displayed. The authors of an editor can either use the class `RootBlock` directly to add the contents of the document to it, or inherit from the `RootBlock` class to provide custom functionality or to override the default behavior. This code sample shows how the block which models a C# compilation unit inherits from the `RootBlock` class:

```
using GuiLabs.Editor.Blocks;

namespace GuiLabs.Editor.CSharp
{
    public class CodeUnitBlock : RootBlock, ICSHarpBlock
    {
        public CodeUnitBlock() : base()
        {
            this.Add(new EmptyNamespaceBlock());
            ...
        }
        ...
    }
}
```

5.6. HContainerBlock, VContainerBlock

Both `HContainerBlock` and `VContainerBlock` are simple containers, which automatically align their child blocks horizontally (`HContainerBlock`) or vertically (`VContainerBlock`). This code sample shows how a `HContainerBlock` can be used to place four blocks beside each other:

```

public partial class TutorialForm : Form
{
    private RootBlock document = new RootBlock();

    private LabelBlock captionCelsius = new LabelBlock("Celsius: ");
    private TextBoxBlock value = new TextBoxBlock();
    private LabelBlock captionFahrenheit =
        new LabelBlock(" Fahrenheit: ");
    private LabelBlock result = new LabelBlock();

    public TutorialForm()
    {
        InitializeComponent();

        viewWindow1.RootBlock = document;

        HContainerBlock row = new HContainerBlock();
        row.Add(captionCelsius, value, captionFahrenheit, result);
        document.Add(row);

        value.TextChanged += value_TextChanged;
    }

    private void value_TextChanged(
        ITextProvider sender,
        string oldText,
        string newText)
    {
        double celsius = 0;
        double.TryParse(newText, out celsius);
        double fahrenheit = celsius * 9 / 5 + 32;
        result.Text = fahrenheit.ToString();
    }
}

```

This sample shows a horizontal line where the user can enter the temperature in centigrade and it is automatically converted to Fahrenheit.

5.7. LinearContainerBlock

Both `HContainerBlock` and `VContainerBlock` are trivial subclasses of the `LinearContainerBlock` class, which provides rich possibilities to align the children horizontally or vertically.

By default, an instance of a `LinearContainerBlock` behaves like `VContainerBlock`, but the orientation can be changed at runtime by setting the `Orientation` property.

5.8. TextBoxBlock

Another important type of block is the `TextBoxBlock` (similar to `TextBox`, `EditText` etc. in common windowing systems). It represents a single text string on the screen, which can be edited by the user.

This example adds a `TextBoxBlock` called “name” to a container block:

```
TextBoxBlock name = new TextBoxBlock();  
this.Add(name);
```

The `TextBoxBlock` class also has a constructor that accepts the initial value of the displayed string. The `Text` property can be read from or written to:

```
name.Text = "Class1";  
MessageBox.Show(name.Text);
```

When the `Text` property is set, the following things happen. If the `TextBoxBlock` belongs to a block tree with a root, the text change is treated as a transaction and is recorded in the Undo/Redo buffer. After the change, the entire tree is redrawn on the screen (if not inside a redraw accumulation context), so that the user notices the change visually. Same happens when the user edits the text through the user interface (it is equivalent to setting the `Text` property programmatically). If the `Root` property of the `TextBoxBlock` object is `null` (the block “hangs in the air”), the change is carried out immediately. Nothing is being redrawn in this case.

Reading the `Text` property has no observable side effects.

The `TextBoxBlock` class also provides the `TextChanged` event, which is raised when the text has actually been changed (by the user or programmatically), and not when the text change transaction is registered with the `ActionManager`. This is important, because in a transaction context the change won’t be visible immediately:

```

[TestMethod]
public void TestSetTextInTransaction()
{
    TextBoxBlock name = new TextBoxBlock();
    Root.Add(name);

    name.Text = "A"; // takes effect immediately
    Assert(name.Text == "A");

    using (Transaction t = new Transaction(Root))
    {
        // takes effect only after
        // the transaction is committed (disposed)
        name.Text = "B";
        Assert(name.Text == "A");
    }

    // "B", because the transaction was committed
    // after exiting the using statement
    Assert(name.Text == "B");

    name.Delete();
}

```

It is important to know that the `TextBoxBlock` uses a `TextBox` control for the physical representation. The special `MyTextBox` property provides a direct reference to this `TextBox` control. One could use this, for example, to set the default minimum width of the text box in pixels:

```

name.MyTextBox.MinWidth = 30;

```

Other than that, the `TextBox` control provides useful API to work with text, selection and the caret position: `Text`, `TextBoxChangePending`, `TextChanged`, `CaretIsAtBeginning`, `CaretIsAtEnd`, `CaretPosition`, `CaretPositionChanged`, `SelectionStart`, `SelectionEnd`, `SelectionLength`, `SelectionText`, `TextBeforeCaret`, `TextBeforeSelection`, `TextAfterCaret`, `TextAfterSelection`, `SetCaretToBeginning`, `SetCaretToEnd`, etc.

5.9. TextBoxBlockWithCompletion

The `TextBoxBlockWithCompletion` class inherits from the `TextBoxBlock` class and provides additional functionality to show the completion list. A completion list is a drop-down list box which appears near the text box when some event occurs (the user is typing something or pressing some key etc.). This can be used for different purposes in a target

editor – for example, to show a list of methods when the user inputs a type name followed by a dot.

The `TextBoxBlockWithCompletion` provides the `Completion` property of type `CompletionFunctionality`, which allows to control, show or hide the completion list. Besides, the `TryShowCompletionList` method is a shortcut to quickly show the completion list and to preselect an item based on the entered prefix.

5.10. LabelBlock

To model non-editable text on the screen, a `LabelBlock` can be used:

```
this.Add(new LabelBlock("("));
```

The `LabelBlock` class provides three constructors – an empty one, a constructor that accepts a default string value and a constructor that accepts an `ITextProvider`.

One can get and set the `Text` of the label using the `Text` property:

```
LabelBlock info = new LabelBlock("Helpful hint!");  
info.Text = "Another hint!";
```

The user of the editor cannot change the text directly. However, it is possible to allow the user to select the label (to put focus on it). To allow user to focus the label, set the `Focusable` property to true. When a focusable label is focused, the user can also delete it by pressing the `[Delete]` key.

When a `LabelBlock` is initialized using an `ITextProvider` object, it is bound to automatically display the current value of the text provider object. When the value changes, the `Text` of the `LabelBlock` is changed automatically to always stay in sync with the text provider.

5.11. UniversalBlock

The UniversalBlock is an important example of a ContainerBlock. This is a specialized container that consists of two compartments – a horizontal line compartment above and a vertical “members list” compartment below:

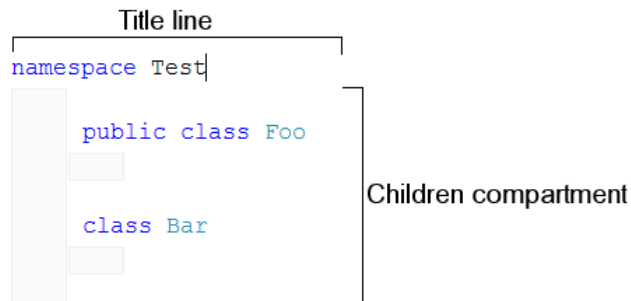


Figure 20 - an example of a UniversalBlock

One can put arbitrary blocks in these two compartments. For example, a LabelBlock and a TextBoxBlock could be placed into the horizontal compartment to form a “header” of the entire container. At the same time real child blocks could be displayed in the vertical compartment, which is normally indented to the right on the screen.

```
this.NameBlock = new TextBoxBlock();
this.HMembers.Add(new LabelBlock("namespace "));
this.HMembers.Add(this.NameBlock);
this.VMembers.Add(new EmptyUsingNamespaceBlock());
```

The UniversalBlock has two main properties which correspond to the two compartments: HMembers of type HContainerBlock and VMembers of type VContainerBlock. Additionally, the UniversalBlock contains a collapse/expand button, which can show or hide the horizontal compartment. The UniversalBlock itself is implemented as a VContainerBlock containing HMembers and VMembers.

5.12. ButtonBlock

The ButtonBlock models a usual button from common graphics interface libraries. The user can push the button with the mouse or by pressing [Enter] or [Space] when the button has focus. The button can contain arbitrary content, most often some text, a picture

or a combination of both. The overloaded constructors of the `ButtonBlock` class allow to create a button with text, with a picture or with both.

The following sample code shows and hides a label, when the user clicks on a button:

```
public class TutorialRootBlock4 : RootBlock
{
    public TutorialRootBlock4() : base()
    {
        button = new ButtonBlock("Hide the text");
        text = new TextBoxBlock();
        this.Add(button);
        this.Add(text);

        button.Pushed += button_Pushed;
    }

    void button_Pushed()
    {
        text.Visible = !text.Visible;
        button.Text = "Text.Visible = "
            + text.Visible.ToString();
    }

    TextBoxBlock text;
    ButtonBlock button;
}
```

5.13. EmptyBlock

`EmptyBlock` models the insertion space between two blocks, where new blocks can be inserted. It looks like an empty text box where the user can place focus and start typing. When the text is entered, a drop-down completion list is shown offering what to insert instead of the current empty block. The class `EmptyBlock` inherits from `TextBoxBlockWithCompletion` and that is why working with the completion list is absolutely the same as by `TextBoxBlockWithCompletion`.

In the following code sample a special empty block is defined, which allows inserting `TutorialUniversalBlock` and another copy of empty block after itself. This copy is necessary so that an empty block always separates two universal blocks.

```

using GuiLabs.Editor.Blocks;

namespace GuiLabs.Editor.Sample
{
    public class TutorialEmptyBlock : EmptyBlock
    {
        protected override void FillItems()
        {
            Completion.AddCreateBlocksItem
                <TutorialUniversalBlock, TutorialEmptyBlock>
                ("universal");
        }
    }
}

```

To define the contents for the completion list, use the `Completion` property, just like with usual `TextBoxBlockWithCompletion`.

The most common completion list item for an empty block is: “insert block(s) after the current empty block”. This is represented by an object of type `CreateBlocksItem`:

```

CreateBlocksItem newItem = new CreateBlocksItem(
    "universal",
    new Type[]
        {typeof(TutorialUniversalBlock), typeof(TutorialEmptyBlock)},
    this);
this.Completion.Items.Add(newItem);

```

This code inserts a new item in the completion list, which is shown as “universal”. When the user selects this entry from the list, two new blocks are inserted after the current empty block, of types `TutorialUniversalBlock` and `TutorialEmptyBlock`.

There is a shortcut for this functionality – the method `AddCreateBlocksItem` of the `CompletionFunctionality` class:

```

Completion.AddCreateBlocksItem
    <TutorialUniversalBlock, TutorialEmptyBlock>
    ("universal");

```

This does basically the same as above, but is shorter and easier to read.

One shouldn’t mix the completion list items to create blocks with the blocks themselves. The completion list items are objects of type `CompletionListItem`, which can execute any functionality when clicked. For example, an object of type `CreateBlocksItem` creates new blocks and adds them to the tree. Thus, `CreateBlocksItem` is a sort of a prototype of

blocks, it contains instructions on how to create new blocks and where to place them. That's why it doesn't store any concrete blocks, but rather their types – when clicked, it creates new instances of those types and adds them after the current empty block.

Beside `CreateBlocksItem`, which inserts new blocks after the current `EmptyBlock`, there is also `ReplaceBlocksItem`, which replaces the current `EmptyBlock` with new blocks:

```
Completion.AddReplaceBlocksItem<TutorialUniversalBlock>("universal2");
```

This code creates a completion list item to replace an `EmptyBlock` with an instance of `TutorialUniversalBlock`.

Other sorts of completion list items can be defined by the users of the framework. It is sufficient to inherit from the `CompletionListItem` class. For example, `CreateBlocksItem` and `ReplaceBlocksItem` themselves inherit from `CompletionListItem`.

6. Implementation of the functionality

6.1. Designing the user interface

6.1.1. Displaying a RootBlock inside a ViewWindow

Given a block tree (the document), it is necessary to show this document to the user. A special Windows Forms control `ViewWindow` (defined in `Core`) takes care of showing any `RootBlock` with its children on the screen. `ViewWindow` embeds the world of blocks into the world of Windows Forms controls and is an important element of the user interface. `ViewWindow` is based on the `DrawWindow` class from the `Canvas` library.

Suppose we have already defined all blocks including the class `MyRootBlock`. We also have a Windows Form with an empty `ViewWindow` control. Now we'd like to display the instance of `MyRootBlock` class in the `ViewWindow` control. The `ViewWindow` class provides a special property for this, which is called `RootBlock`:

```
public frmMain()
{
    InitializeComponent();
    // display the block tree in the ViewWindow control
    viewWindow1.RootBlock = Document;
}

// create an instance of RootBlock and save it in the Document property
private MyRootBlock mDocument = new MyRootBlock();
public MyRootBlock Document
{
    get { return mDocument; }
    set { mDocument = value; }
}
```

When the property `ViewWindow.RootBlock` is set, a complex binding process takes place, which subscribes to the events of the `RootBlock` and initiates the communication. Mouse and keyboard events from the `ViewWindow` are being re-routed to the `RootBlock`, drawing of the `ViewWindow` contents is being delegated to the `RootBlock` as well. When the `RootBlock` needs to be scrolled, `ViewWindow` manages the scrollbars. All this happens transparent for the user.

6.2. Focus

Blocks can be focused. This means, each instance of the class `RootBlock` at runtime can have a reference to a single currently active block (which is visually highlighted and receives the keyboard input).

Not all blocks can be focused. For example, a `ContainerBlock` which only exists to group other blocks, cannot be focused by default. A `LabelBlock` isn't normally focused as well. The `CanGetFocus` property of such blocks returns `false`. If it is necessary to make a block focusable, set the `Focusable` property to `true`.

The user can focus a block by clicking on its control with the mouse. When the user clicks on a non-focusable block, the mouse event is routed to the nearest focusable parent up the parent chain. Normally the user can also move focus with the keyboard arrow keys. This functionality is automatically implemented for most basic blocks. A depth-first traversal of the tree structure induces the order of blocks in which they are traversed. Thus, all of the blocks in any tree can be put in a numbered sequence with the following property: each block precedes all its children and the next sibling comes after all the children of the current block. We'll call this order the block order.

The classes `Block` and `ContainerBlock` (the most important base blocks of the framework) provide useful API to control the focus. Almost all of these methods can be overridden if necessary.

- `void SetFocus()` - gives the block focus, if possible. The screen is updated automatically, if the block has a root block (is within a tree). Blocks, which are currently invisible or disabled, cannot be focused (for example, blocks from the collapsed compartment of a `UniversalBlock` cannot be focused until the compartment is expanded again). The method calls `CanGetFocus` to determine if the block can currently receive focus.
- `void SetDefaultFocus()` - sets focus to the container block the first time after its creation. Override this method to give the focus to some part of a complex block right after its creation. This method is called automatically for blocks that are added to the tree.
- `void RemoveFocus()` - moves the focus somewhere else away from the current block, if possible. If the current block wasn't focused, the method has no effect. `RemoveFocus` is normally used before deleting or hiding blocks so that the focus remains on some existing neighbor block. The current implementation doesn't guarantee, which block receives focus and this behavior is deliberately not specified.
- `bool CanGetFocus { get; }` - determines, if a block can currently receive focus. A block can receive focus if it is visible, enabled, contained within a tree (Root is not null), not within a collapsed or hidden subtree and its `Focusable` property is set to true.
- `IBlock FindFirstFocusableChild()` - returns the first focusable child block of the current block, if such a block exists, null otherwise. Searches depth-first recursively, without considering the current block. Can only be non-null for containers.
- `IBlock FindLastFocusableChild()` - returns the last focusable child block of the current block, if such a block exists, null otherwise. Searches depth-first recursively from bottom to top, without considering the current block. Can only be non-null for containers.

- `IBlock FindFirstFocusableBlock()` – returns the first focusable child block of the current block, if such a block exists, or itself if focusable, null otherwise. Searches depth-first recursively, starts with the current block. Functionality is similar to `FindFirstFocusableChild`, but the current block is tested too.
- `IBlock FindLastFocusableBlock()` – returns the last focusable child and the current block, if no focusable children were found. If no blocks were focusable, returns null. Works similar to `FindLastFocusableChild`.
- `IBlock FindPrevFocusableBlock()` – returns the previous focusable block in the block order, but stops searching and returns null after all the siblings have been searched. Thus, it will never return the blocks parent, but will stop and return null.
- `IBlock FindNextFocusableBlock()` – returns the next focusable block in the search order, but only searches the siblings.
- `IBlock FindPrevFocusableBlockInChain()` – returns the previous focusable block in the block order. Searches up to the `RootBlock` and works bottom to top and depth-first.
- `IBlock FindNextFocusableBlockInChain()` – this is the natural enumerating method, that returns the next block in the block order, up to the last block of the tree.
- `bool SetCursorToTheBeginning()` – a more fine-granular version of `SetFocus`, that differentiates between the start and the end of a block. For example, for a `TextBoxBlock` it would set focus to the block and position the caret at its beginning. For less granular block where there is no difference, the entire block is focused (as usual). For a container block, calls `SetCursorToTheBeginning` recursively for the first focusable child.
- `bool SetCursorToTheEnd()` – just like `SetCursorToTheBeginning`, but attempts to set the focus to the end of the block. For a container block, calls `SetCursorToTheEnd` recursively for the last focusable child.

6.3. Events and user interaction

To allow full user interaction, the author of a structured editor has to implement the reaction of the editor to the user input actions. For example, the mouse clicks on a block have to be processed. Blocks and controls provide a set of events which are raised when the user input has to be processed.

6.3.1. Block.KeyDown event

All Blocks define the `KeyDown` event. This event is raised for the currently focused block when the user presses a keyboard key.

We illustrate this on the example of the `foreach`-block in the C# editor. `ForeachBlock` models the `foreach` statement of the C# language and is represented by a `UniversalBlock`, which has three text boxes in its horizontal (`HMembers`) compartment. We'd like to enable the user to switch focus between these three text boxes using the `[Tab]` key. This simplified code shows how to process the `keydown` event:

```

public class ForeachBlock : ControlStructureBlock
{
    private ExpressionBlock IteratorType = new ExpressionBlock();
    private ExpressionBlock IteratorName = new ExpressionBlock();
    private ExpressionBlock CollectionName = new ExpressionBlock();

    public ForeachBlock() : base("foreach")
    {
        this.HMembers.Add(new LabelBlock("("));
        this.HMembers.Add(IteratorType);
        this.HMembers.Add(new LabelBlock(" "));
        this.HMembers.Add(IteratorName);
        this.HMembers.Add(new LabelBlock(" in "));
        this.HMembers.Add(CollectionName);
        this.HMembers.Add(new LabelBlock(")"));

        IteratorType.KeyDown += IteratorType_KeyDown;
        IteratorName.KeyDown += IteratorName_KeyDown;
        CollectionName.KeyDown += CollectionName_KeyDown;
    }

    void IteratorType_KeyDown(IBlock Block, KeyEventArgs e)
    {
        if (e.KeyCode == System.Windows.Forms.Keys.Tab)
        {
            IteratorName.SetFocus();
        }
    }

    ...
}

```

6.3.2. Block.ParentChanged event

ParentChanged is another event which all of the blocks possess. It is raised when the block gets a new parent, that is when it is first added to some ContainerBlock or moved to another ContainerBlock. When the block is deleted, its ParentChanged event is raised as well.

6.3.3. Children.CollectionChanged

The IChildrenList interface represents an observable collection. It provides several events which notify the clients when the children collection has changed. The most general event is the CollectionChanged. It is raised when a block has been inserted, deleted or replaced. In case when several blocks are added or removed at the same time, the event is raised for each change separately.

As an example we consider a hypothetical feature of the C# editor which visually displays a total number of methods for each class. The following code solves the problem:

```

public class ClassBlockWithMethodCounting : ClassBlock
{
    public ClassBlockWithMethodCounting() : base()
    {
        // prepare a new horizontal container for our "status bar"
        HContainerBlock infoLine = new HContainerBlock();

        // add three components to the infoLine,
        // the middle one will be changed dynamically
        infoLine.Add(new LabelBlock("This class has "));
        infoLine.Add(statistics);
        infoLine.Add(new LabelBlock(" methods."));

        // add the line to the class block
        this.VMembers.AddToBeginning(infoLine);

        // here we subscribe to the CollectionChanged event
        this.VMembers.Children.CollectionChanged += Children_Changed;
    }

    private LabelBlock statistics = new LabelBlock("0");

    void Children_Changed()
    {
        // get the list of all methods in this class
        ReadOnlyCollection<MethodBlock> methods =
            this.VMembers.FindChildren<MethodBlock>();

        // update the text with the current value
        statistics.Text = methods.Count.ToString();
    }
}

```

6.3.4. RootBlock.ActiveBlockChanged

Each `RootBlock` provides a special `ActiveBlockChanged` event which is raised after a new block is focused. This is useful for example, when it is necessary to display some information about the currently focused block (for example, context help). This code sample shows the type of the currently focused block:

```

public partial class TutorialForm : Form
{
    private TutorialRootBlock document = new TutorialRootBlock();
    private LabelBlock info = new LabelBlock();

    public TutorialForm()
    {
        InitializeComponent();

        viewWindow1.RootBlock = document;
        document.Children.Add(info);
        document.ActiveBlockChanged += document_ActiveBlockChanged;
    }

    void document_ActiveBlockChanged(
        IBlock oldFocused,
        IBlock newFocused)
    {
        if (newFocused != null)
        {
            info.Text = "Focused: " + newFocused.ToString();
        }
    }
}

```

The property `RootBlock.ActiveBlock` returns the currently focused block of the tree.

6.4. Actions

Each operation on the block data structure (adding, renaming, deleting etc.) can be recorded as an undoable action. Every change is explicitly separated into two steps: preparing an action and (possibly much later) running it.

This is an implementation of the Command design pattern (see [GoF]). But we use a slightly different terminology: the GoF command (the encapsulated operation) is called an *Action* in the editor framework, whereas each user input unit (an encapsulated menu item or a toolbar item) is called a *Command*. Thus, commands invoke actions, but remain independent of actions. Commands pertain to the UI and actions are UI independent and only operate on the data structure.

This design pattern demonstrates the principle of delayed execution, which was probably inspired by the functional programming languages, where the code can be treated as data and executed in a lazy manner, upon request.

Each action is represented by an object of type `IAction`. The `IAction` interface has two important methods: `Execute()` and `UnExecute()`. The `Execute()` method actually runs the encapsulated action and `UnExecute()` reverses it back.

6.4.1. History of executed actions

When an action is executed, it is added to the list of all executed actions – the Undo/Redo buffer, or action “history”. This buffer is represented in memory as a doubly linked list.

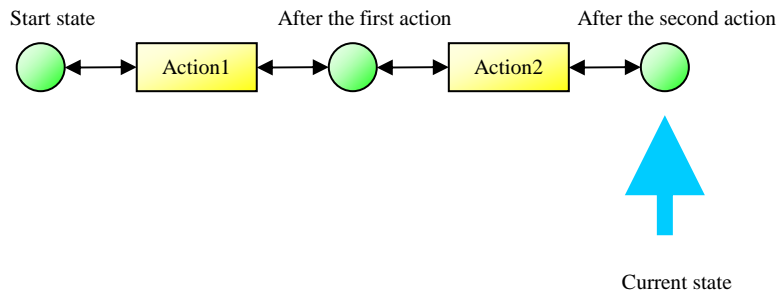


Figure 21 - sample history of actions (two actions have been executed so far)

The green circles represent states of the data structure and the yellow rectangles represent actions that transition between the two states. The blue arrow shows the current state. The objects that represent states in the list carry no information and are only used to better model the action history.

6.4.2. Undo/Redo

Each action can not only be executed, but also can be undone. When an action is undone, it is not deleted from the list, but the pointer to the current state is shifted to the left.

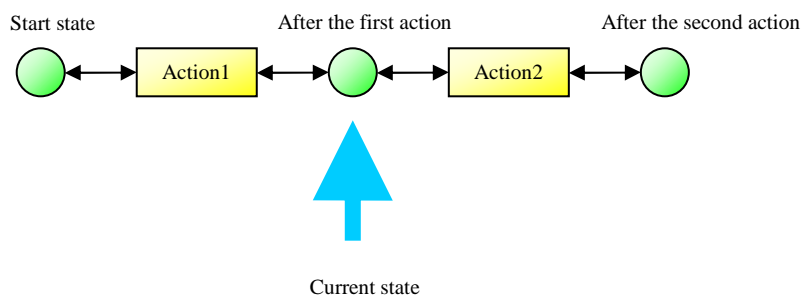


Figure 22 - Action2 was undone

When the pointer moves to the left, it encounters the Action2 and calls its `UnExecute()` method. The “Redo” operation moves the pointer to the right and calls the `Execute()` method on the next encountered action.

The “Undo” operation is accessible when there are actions to the left of the pointer. The “Redo” operation is accessible when there are actions to the right of the pointer. When there are actions to the right of the pointer that can be redone and the user records a totally new action, all earlier actions to the right of the pointer are discarded (the user chose not to redo them but to record a new action manually). This new action is placed to the end of the list and the pointer is set to point to it:

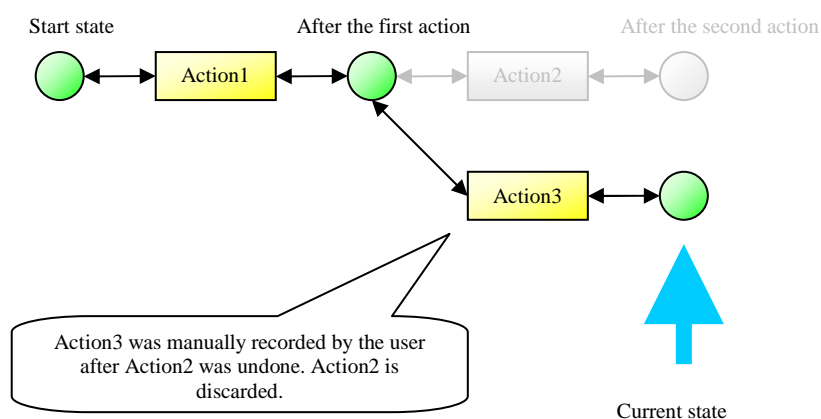


Figure 23 – Redo buffer is discarded

This implementation is a little bit different from the traditional implementation with two stacks – one for undo and one for redo. It is more flexible and allows for providing more complicated redo scenarios in the future, such as preserving the redo actions in a tree and allowing the user to choose from several possible actions to redo. Thus the user could try doing scenario 1, undo, try scenario 2, undo and then when redoing, not only scenario 2 would be possible, but the choice between scenario 1 and scenario 2.

6.4.3. Implementation

The `ActionManager` class implements the Undo/Redo buffer of a document (`RootBlock`). Each `RootBlock` has a reference to its dedicated `ActionManager` via the `ActionManager` property. When a new `RootBlock` is created, its `ActionManager` is created automatically. The `ActionManager` has the following members:

<code>bool CanUndo { get; }</code>	Returns true if there is an undoable action in the Undo/Redo buffer.
<code>bool CanRedo { get; }</code>	Returns true if there is a redoable action in the Undo/Redo buffer.
<code>void Undo();</code>	Undoes the last action and decrements the pointer to the current state.
<code>void Redo();</code>	Redoes the last undone action and increments the internal current state pointer.
<code>void RecordAction (IAction existingAction);</code>	Inserts a new Action to the undo buffer. If there is a transaction which is currently being recorded, the action is added to the transaction, otherwise the Action is executed.
<code>IActionHistory History { get; set; }</code>	A reference to the doubly linked list with the actions and states.

With the help of this functionality one can easily traverse the saved actions, undo, redo and record new actions. The following code undoes and redoes the last action:

```
rootBlock.ActionManager.Undo();
rootBlock.ActionManager.Redo();
```

This operation can be called from every place in the program, including the user interface. That is how one can implement the Undo and Redo menus of the application.

6.4.4. Kinds of actions

The most important classes of the framework, which represent actions, inherit from the `RootBlockAction` class. They are:

- `AddBlocksAction` – inserts new block(s) after specified block or to the end of the blocks `Children` list.
- `PrependBlocksAction` – prepends block(s) before specified block.

- `ReplaceBlocksAction` – replaces the given block with other block(s).
- `RemoveBlocksAction` – removes one or more blocks from the block tree.
- `RenameItemAction` – renames a text item (mostly the `TextBoxBlock`).

These actions accept already existing blocks as parameters and do not create blocks themselves. The blocks must be created externally and passed to the action.

6.4.5. Creating the action explicitly using the constructor

One can directly construct the necessary action object using one of the predefined constructors. The following code inserts a new `TextBoxBlock` after the current block, so that the action is registered with the `ActionManager` and executed:

```
AddBlocksAction action = new AddBlocksAction(this.Parent, this);
action.PrepareBlocks(new TextBoxBlock());
this.ActionManager.RecordAction(action);
```

This way is the most flexible and allows full control to configure and prepare the action. The disadvantage of this way is that it is mostly a lot of code – instantiating, configuring and running the action.

6.4.6. Methods on the Block class

The `Block` class provides several shortcut methods to perform operations on the data structure, which automatically prepare and execute necessary actions. The following code does absolutely the same as the previous code sample:

```
this.AppendBlocks(new TextBoxBlock());
```

This version is much more expressive and easier to use. The `AppendBlocks` method takes care of preparing and running the action for us. If no `ActionManager` is available for the current block (when it is not within any tree), the action is executed directly; i.e. the block is added immediately.

6.4.7. ActionFactory

The `ActionFactory` class provides shortcut methods which create and returns actions without the necessity to explicitly create the action object. Let's rewrite the code above using the `ActionFactory` class:

```
AddBlocksAction action = ActionFactory.AddBlock
    (this, new TextBoxBlock());
this.ActionManager.RecordAction(action);
```

This way is useful when the caller wants to have access to the action before it goes to the `ActionManager`. It is as flexible as directly creating an action but is still less verbose when compared to explicitly creating an action.

6.4.1. Transactions

Transactions represent a powerful way to group actions into atomic operations that can be treated as a whole (design pattern Command). Here's an example of a using a transaction – here a Move operation is implemented as a composition of Delete and Add operations:

```
public static void MoveBlock(ContainerBlock newParent, Block blockToMove)
{
    using (Transaction t = new Transaction(newParent.Root))
    {
        blockToMove.Delete();
        newParent.AppendBlocks(blockToMove);
    }
}
```

Transactions are undone and redone as a single action. When a transaction is created, the `ActionManager` switches into recording state – instead of running the actions directly, it starts accumulating them in a transaction that is being opened. Other transactions can be registered this way as well. As soon as the last top-level transaction goes off the stack (its `Dispose()` method is called), the `ActionManager` goes back into the execution state and runs the entire complex transaction.

After an action is executed, the view where the blocks are drawn (`ViewWindow`) is repainted automatically, so that the user can see the changes on the screen. To prevent multiple repaintings during a transaction, all painting inside the transaction scope is suppressed and the entire `RootBlock` is repainted only once after the transaction completes. If transactions are nested, only the top-level transaction causes a repaint when finished. To prohibit repainting after a transaction is done, add the following code to the transaction:

```
t.AccumulatingAction.ShouldRedrawWhenDone = false;
```

6.4.2. UndoBufferChanged Event

The `ActionManager` class provides an `UndoBufferChanged` event. This event is raised when the Undo/Redo buffer is changed – when adding new actions, undoing or redoing.

The following code sample demonstrates how one can automatically react to any change of the document (here we automatically call a hypothetical `AutoSave()` method):

```
public partial class TutorialForm : Form
{
    private TutorialRootBlock document = new TutorialRootBlock();

    public TutorialForm()
    {
        InitializeComponent();

        viewWindow1.RootBlock = document;
        document.ActionManager.UndoBufferChanged +=
            ActionManager_UndoBufferChanged;
    }

    void ActionManager_UndoBufferChanged()
    {
        document.AutoSave();
    }
}
```

6.5. Controls

It is important to understand that blocks represent a logical data structure which is used to model a language construct. Blocks do not store any visualization information directly. Instead, controls are used to represent blocks on the screen.

The class `Block` has the property `MyControl` of type `Control`. This is a block's reference to the corresponding control object, which stores information important for visualization and interaction. Each block knows and can access and work with its own control.

Blocks are also responsible for creating and managing their controls as well.

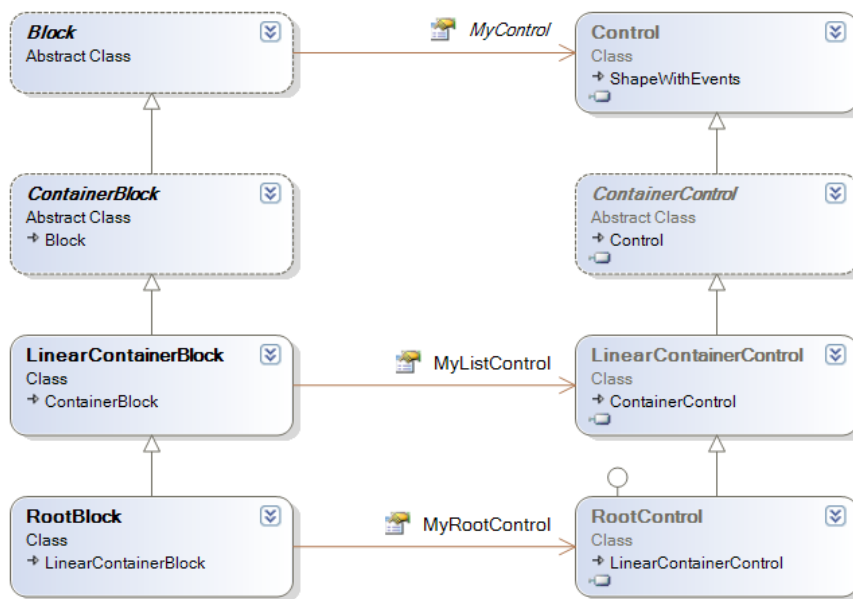


Figure 24 - parallel hierarchies of blocks and controls

The following sample shows how to set the minimal width of a text box by accessing the `TextBox` control of the `TextBoxBlock` class:

```

public class TutorialRootBlock3 : RootBlock
{
    public TutorialRootBlock3() : base()
    {
        HContainerBlock con = new HContainerBlock();
        this.Add(con);

        TextBoxBlock text = new TextBoxBlock();
        con.Add(text);
        con.Add(new LabelBlock("// end of the text box"));

        text.MyTextBox.MinWidth = 40;
    }
}
  
```

The property `MyTextBox` is similar to `MyControl`, but it is more exactly typed: it returns an object of type `TextBox` instead of returning an object of type `Control`. Thus the users do not need to cast `MyControl` to the `TextBox` type.

Each `Control` provides a rich choice of events, which are raised by mouse or keyboard input. Here is an example where the background color of the document is randomly changed when the user clicks on a label. The current mouse cursor coordinates are being shown as well:

```

public class TutorialRootBlock5 : RootBlock
{
    public TutorialRootBlock5() : base()
    {
        label = new LabelBlock("Click me");
        mouseCoords = new LabelBlock();
        this.Add(label, mouseCoords);

        label.MyControl.MouseDown += MyControl_MouseDown;
        this.MyControl.MouseMove += delegate
            (MouseEventArgsWithKeys MouseInfo)
        {
            mouseCoords.Text = MouseInfo.X + "; " + MouseInfo.Y;
        };

        this.MyControl.Style.FillStyleInfo.Mode =
            FillMode.HorizontalGradient;
    }

    LabelBlock label, mouseCoords;

    void MyControl_MouseDown(MouseEventArgsWithKeys MouseInfo)
    {
        this.MyControl.Style.FillStyleInfo.FillColor
            = GetRandomColor();
        this.MyControl.Style.FillStyleInfo.GradientColor
            = GetRandomColor();
        this.MyControl.Redraw();
        label.Text = "Thank you!";
    }
}

```

6.6. VisibilityConditions

It is possible to prevent some completion list items from being shown under some conditions. For instance, when a block can only be inserted once, the menu item should check if such a block already exists before being shown in the completion list.

A mechanism which determines if a `CompletionListItem` should be shown or not relies on a list of so called visibility conditions. Each condition is an object of type `ICondition`:

```

public interface ICondition
{
    bool IsTrue();
}

```

Each `CompletionListItem` has a list of such conditions. The item is only visible when all of the conditions are true (for each `ICondition` its `IsTrue()` method returns true).

At the beginning the list is empty and hence all of the conditions are true – the item is visible.

The following sample code adds a visibility constraint to the “text” item, to allow not more than 3 TextBoxBlocks within the entire tree:

```
public class TutorialEmptyBlock : EmptyBlock
{
    protected override void FillItems()
    {
        CreateBlocksItem createText =
            Completion.AddCreateBlocksItem<TextBoxBlock>("text");
        MoreThan3TextBlocksCondition createTextVisible =
            new MoreThan3TextBlocksCondition(this);
        createText.VisibilityConditions.Add(createTextVisible);
    }

    private class MoreThan3TextBlocksCondition : ICondition
    {
        public MoreThan3TextBlocksCondition(EmptyBlock parent)
        {
            Parent = parent;
        }

        private EmptyBlock Parent;

        public bool IsTrue()
        {
            ReadOnlyCollection<TextBoxBlock> textBoxes =
                Parent.Root.FindChildrenRecursive<TextBoxBlock>();
            return textBoxes.Count < 3;
        }
    }
}
```

7. Implementing the C# editor

We will now describe how the structured editor framework was used to create a source code editor for the C# programming language.

7.1. The project structure

The editor component could be used either in a standalone application or hosted inside an IDE. Some features of the editor (e.g. code completion) will only be available when hosted within some IDE. At the same time, the editor component is generic and does not depend on any specific IDE. Instead of hard-coding the dependency, the editor provides flexible extension points where the host could plug in and provide necessary information to integrate with the editor.

The editor itself is implemented as a library of blocks and additional functionality named `CSharpBlocks` and packaged in a .NET Framework .dll assembly (`GuiLabs.Editors.CSharp.dll`). The editor assembly references the four framework assemblies: `Utils`, `Canvas`, `Controls` and `Core`. It is independent of any other IDE specific assemblies.

7.2. Defining data structures (blocks)

The core of the editor are classes that model C# language constructs: namespaces, types, members, parameters, access modifiers, control structures and some additional auxiliary blocks which do not represent any language constructs themselves.

7.2.1. CodeUnitBlock

A C# code compile unit is represented by the `CodeUnitBlock` class, which inherits from `RootBlock`. Initially, a `CodeUnitBlock` object contains one instance of the `EmptyNamespaceBlock` class.

7.2.2. EmptyNamespaceBlock

An `EmptyNamespaceBlock` doesn't model any language concept directly; it represents an insertion space between two namespaces or type declarations. This class inherits from `EmptyBlock` and defines the `FillItems()` method, which serves to fill the completion drop-down list:

```
protected virtual void FillItems()
{
    ReplaceBlocksItem usingItem =
        Completion.AddReplaceBlocksItem
            <UsingBlock, EmptyNamespaceBlock>("using");
    usingItem.VisibilityConditions.Add
    (
        new DelegateCondition( delegate
            { return this.Prev == null; } )
    );

    AddItem<NamespaceBlock>("namespace");
    AddItem<ClassBlock>("class");
    AddItem<StructBlock>("struct");
    AddItem<InterfaceBlock>("interface");
    AddItem<EnumBlock>("enum");
    AddItem<DelegateBlock>("delegate");

    AddEmptyItem("public");
    AddEmptyItem("internal");
    AddEmptyItem("abstract");
    AddEmptyItem("sealed");
    AddEmptyItem("static");
    AddEmptyItem("partial");
}
```

The `usingItem` (which appears as the word “using” in the completion list) is an item of the completion list which (when selected from the drop-down) replaces the current empty block with an instance of `UsingBlock` followed by a new instance of

EmptyNamespaceBlock. It is only visible in the completion list when the EmptyNamespaceBlock is the first block within its parent. This assures that the using declaration can only be inserted at the top of the CodeUnitBlock or at the top of a namespace.

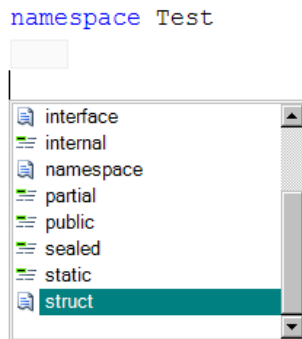


Figure 25 - completion list of an EmptyNamespaceBlock

To enforce the constraint that the completion list item should only be visible within the first EmptyNamespaceBlock, a so-called VisibilityCondition is added to the item's VisibilityConditions collection. For an item to be physically shown in the completion list, all of the visibility conditions must be true at the moment when the drop-down list is displayed. A DelegateCondition means that every boolean method can be used. In this case, we use an anonymous delegate that tests if the current block is the first one in its parent's children list.

Anonymous delegates are a feature introduced in C# 2.0 which implements closures in the language. Thanks to the closure concept, "this" in our code is captured and bound to the EmptyNamespaceBlock instance.

Unfortunately, features of C# 2.0 are not supported by the structured editor itself (at the moment of this writing), but at least they are used extensively in its own source code.

A line

```
AddItem<NamespaceBlock>("namespace");
```

basically defines an item titled "namespace" which creates a new instance of a NamespaceBlock and adds it after the current EmptyNamespaceBlock. It also inserts a new EmptyNamespaceBlock, so that there is always exactly one EmptyNamespaceBlock instance between two NamespaceBlock objects.

There are two main ways, how a new type declaration could be created using `EmptyNamespaceBlock`'s completion list. The first one is straightforward and is based by explicitly selecting the appropriate item ("class", "interface", etc.) from the completion list.

The other way is incremental and is very similar to how we would type it in a plain text editor. For example, one would type "public class Test {" to create a new class in a text editor. Exactly this is supported by the structured editor as well.

A line

```
AddEmptyItem("public");
```

defines that the word `public` starts defining a new type declaration at the current position. When the word "public" is selected, another completion list is presented, which provides the user with the exact context-dependent choice of possible keywords:

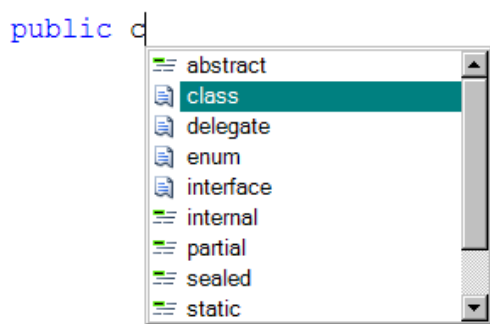


Figure 26 - defining a type incrementally

Some temporary blocks are created during the incremental process, but in the end, a new type declaration block is created (unless the user cancels by undoing or pressing the [Backspace] key).

7.2.3. UsingBlock

Mostly, the class provides methods to manage using and namespace declarations within the code compile unit:

Member type:	Member name:	Description:
Property	UsingSection	Returns the UsingBlock of this CodeUnitBlock, or null if no using

		declaration was defined. Each CodeUnitBlock can contain at most one UsingBlock at the beginning of its children list, which is guaranteed by a special visibility condition on the “using” completion list item.
Method	AddUsing	Inserts a new using directive into the UsingSection. Automatically creates a new UsingSection if this is the first using directive.
Method	AddNamespace	Adds a new namespace declaration to the CodeUnitBlock. Automatically surrounds the namespace declaration by EmptyNamespaceBlock instances.

7.2.4. UsingDirective

UsingDirective represents a single line in the using container and inherits from the TextLine class.

The interesting part is two methods that provide code completion after the user presses a [Dot] key after a part of a namespace name. The first one gets a list of namespace name continuations from the language service, if it is available:

```

public IEnumerable<string> GetNamespaces()
{
    LanguageService service = LanguageService.Get(this);
    if (service != null)
    {
        string namespaceStart = GetNamespaceLeftFromCaret();
        return service.GetNamespaces(namespaceStart);
    }
    return Strings.EmptyArray;
}

```

The other method adds a list of namespace names to the completion list:

```

public void FillCompletionWithNamespaces(
    IEnumerable<string> namespaceNames)
{
    Completion.Items.Clear();
    foreach (string s in namespaceNames)
    {
        Completion.AddTextCompletionItem(s, this, Icons.Namespace);
    }
}

```

7.2.5. CodeBlock

CodeBlock is a base class for many other blocks in the CSharpBlocks library. It inherits from UniversalBlock and adds minimal common functionality to its descendants.

7.2.6. Type declaration blocks

The implementation of blocks that model types is mostly trivial. It is enough to mention that these blocks eventually inherit from the UniversalBlock and present a compartment with child blocks. The following diagram illustrates the inheritance tree of these classes:

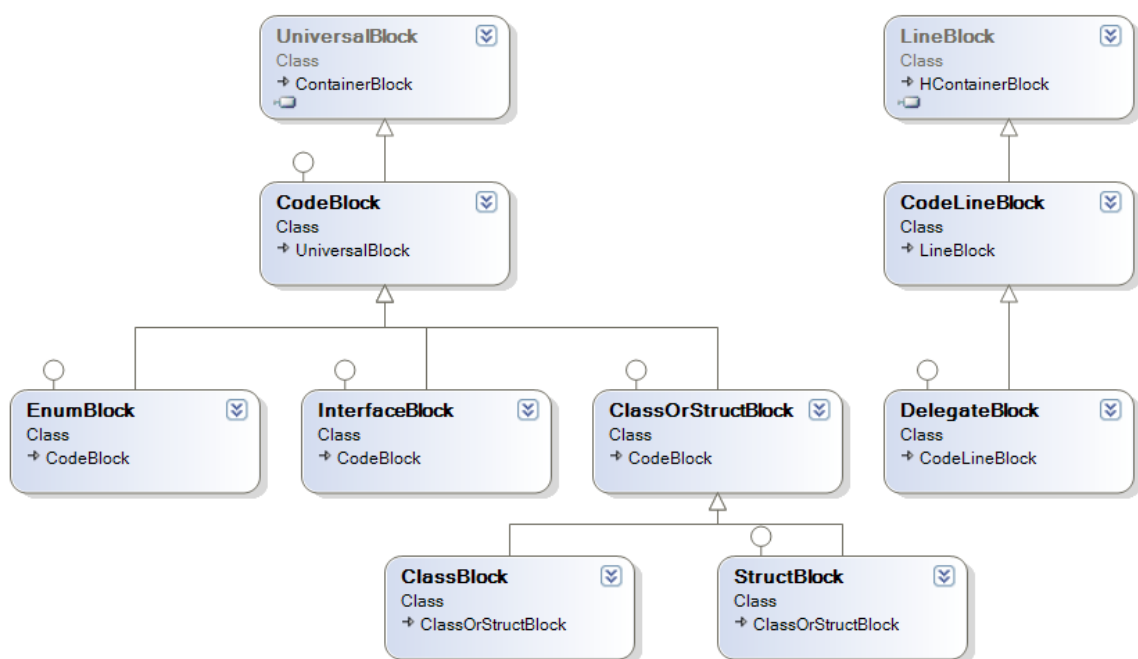


Figure 27 – block classes that model type declarations

Most of these classes define helper functionality to determine the parent class and other useful information:

```

public ClassOrStructBlock ParentClassOrStruct
{
    get { return this.ParentParent as ClassOrStructBlock; }
}

public NamespaceBlock ParentNamespace
{
    get { return this.ParentParent as NamespaceBlock; }
}

public bool IsNested
{
    get { return ParentClassOrStruct != null; }
}

```

Also, they contain child blocks to define the name of the type and access modifiers, represented by `TextBoxBlock` and `ModifierContainer` respectively.

7.2.7. Blocks that represent type members

This class diagram shows the block classes which model type members:

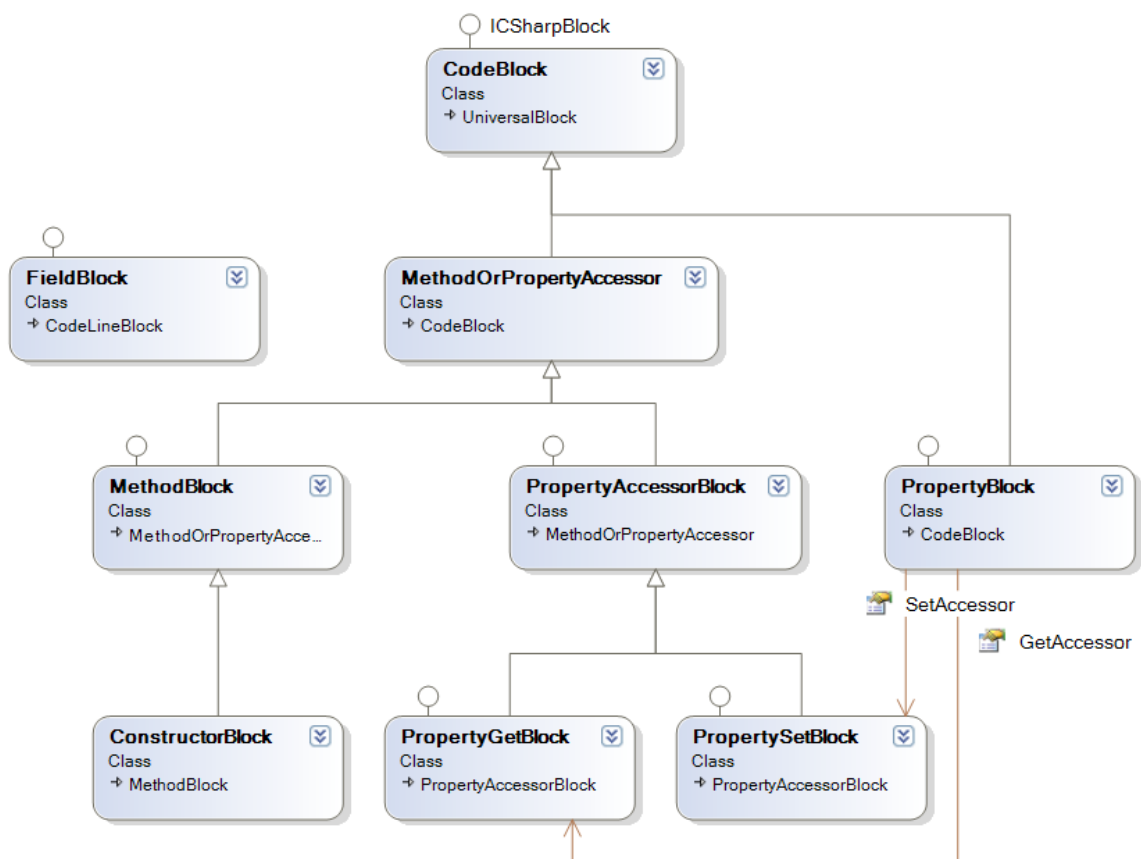


Figure 28 - type member blocks

7.2.8. Control structures

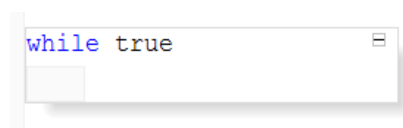
Basic control structure blocks are trivial descendants of the `ControlStructureBlock` class. The `ControlStructureBlock` is a `CodeBlock` with an additional `KeywordLabel` in the title line. Here is, for example, an implementation of the `WhileBlock`, which represents a “while” loop:

```
public class WhileBlock : ControlStructureBlock
{
    public WhileBlock() : base("while")
    {
        ...
        this.HMembers.Add(Condition);
    }

    private ExpressionBlock mCondition = new ExpressionBlock();
    public ExpressionBlock Condition
    {
        get { return mCondition; }
        set { mCondition = value; }
    }

    ...
}
```

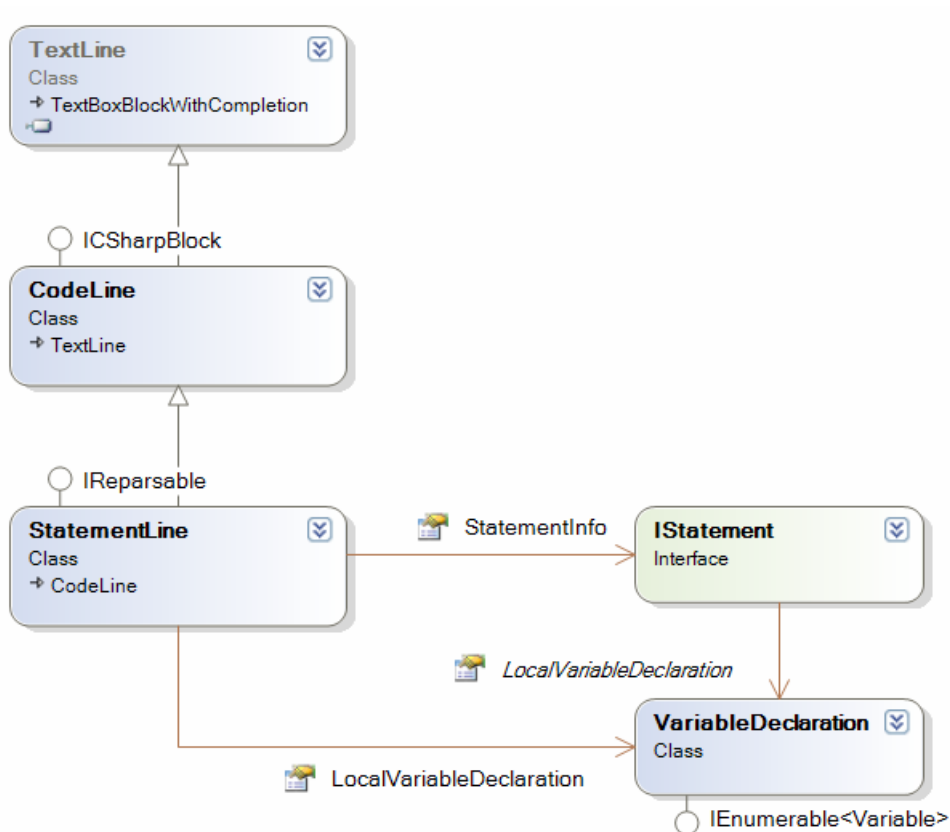
This is for example how the while block looks like:



7.2.9. StatementLine

The inheritance hierarchy of the `StatementLine` class is the following: `StatementLine` ▷ `CodeLine` ▷ `TextLine` ▷ `TextBoxBlockWithCompletion` ▷ `TextBoxBlock` ▷ `Block`. This hierarchy is deliberately so deep, because it uses the concept of double derived classes – one can always insert a new subclass at the necessary position in the hierarchy.

But overall, the `StatementLine` is just a textbox, a line of code, which additionally tracks context information about the statement being edited. For this, it has the `StatementInfo` property of type `IStatement`. All the information is being encapsulated there. Currently, `IStatement` is only used to find out, if the statement actually contains a local variable declaration:



It also declares the `Reparse()` method which updates the statement information using the language service described later:

```

public void Reparse()
{
    LanguageService ls = LanguageService.Get(this);
    if (ls != null && ls.Parser != null)
    {
        StatementInfo = ls.Parser.ParseStatement(this.Text);
    }
    else
    {
        StatementInfo = null;
    }
}

```

7.2.10. ICSharpBlock and Visitors

`ICSharpBlock` is an interface which is implemented by all blocks in the `CSharpBlocks` project. This interface declares the only `AcceptVisitor` method:

```
void AcceptVisitor(IVisitor visitor);
```

which is used in the `Visitor` design pattern to apply some external operation to the implementors of `ICSharpBlock`. This is how an operation encapsulated by `IVisitor` is applied to the entire block family.

The best example for a block visitor is the `PrettyPrinter` class. It contains a number of overloads of the `Visit` method with different parameters. Here is an example for the `for` loop:

```
public override void Visit(ForBlock block)
{
    WriteIndent();
    Write(block.Keyword.Text);
    Write("(");
    Write(block.ForInitializer.Text);
    Write("; ");
    Write(block.ForCondition.Text);
    Write("; ");
    Write(block.ForIncrementStep.Text);
    WriteLine(")");
    StartBlock();
    VisitContainer(block.VMembers);
    EndBlock();
}
```

7.3. Dynamic help

All of the C# blocks define the readonly `HelpStrings` property of type `IEnumerable<string>`. Each block redefines this property to return some text strings to describe the block. These strings can be shown to the user of the editor, for example, when a block is selected.

7.4. LanguageService

The `LanguageService` class is the port through which the structured editor component communicates with its host environment. This is a base class, which defines, but does not implement the functionality, which the editor needs for code completion. When integrated into an IDE or other host environment, the `CodeUnitBlock.LanguageService` property is set. The editor can check, if the `LanguageService` is not null, it means the

host has provided some implementation of the `LanguageService` class and that the editor can use it to show code completion.

The reason why `LanguageService` is a class and not an interface is a conscious design decision. For a good argumentation of this see [FDG], page 77.

One of the applications of the language service is parsing local variable declarations and method parameters. For convenience reasons, the language service has been split into two parts – the general part represented by the `LanguageService` class, and the special parsing part represented by the `ParserService` class. Each `LanguageService` references its `ParserService` part. The `ParserService` class provides two virtual operations:

```
public virtual IStatement ParseStatement(string statementText)
{
    return null;
}

public virtual ParameterList ParseParameters(string parametersText)
{
    return null;
}
```

7.5. ClassNavigator algorithms

Having the language constructs in a well accessible tree structure has many advantages. The algorithms of operating on these constructs become straightforward, because the programmers don't need to concentrate on the parsing part anymore. There are no lines of text, so one can operate on pure concepts.

7.5.1. Determining parent members and types

There are numerous methods to determine, what member/type a given block is in. For example, let's consider the `FindContainingMember` method:

```
public static ContainerBlock FindContainingMember(Block block)
{
    ContainerBlock current = block.Parent;
    while (current != null
        && !(current is MethodBlock)
        && !(current is PropertyBlock))
    {
        current = current.Parent;
    }
    return current;
}
```

7.5.2. Harvesting local variables

“Harvesting” is a process of finding all local variables accessible from the current statement. The title of the algorithm was inspired by the notion of “moving forward and gathering necessary stuff on the way”. The visibility scope rules can be encoded in the following recursive definition:

A variable is visible for a block, if its declaration precedes the block in the same children list or if a variable is visible for it’s parent block.

```
public static IEnumerable<Variable> FindAccessibleVariables(Block point)
{
    List<Variable> foundVariables = new List<Variable>();
    AddAccessibleVariablesStartingFrom(foundVariables, point);
    return foundVariables;
}
```

```

private static void AddAccessibleVariablesStartingFrom(
    List<Variable> vars, Block point)
{
    while (!(point == null
        || point is MethodBlock
        || point is PropertyAccessorBlock))
    {
        while (point != null)
        {
            AddAccessibleVariableDeclaration(vars, point);

            // jump out when we reached the first statement
            // (Dijkstra's loop-and-a-half problem)
            if (point.Prev == null) break;
            point = point.Prev;
        }

        // jump to the parent statement and continue search from here
        point = point.ParentParent;
    }
}

private static void AddAccessibleVariableDeclaration(
    List<Variable> vars, Block node)
{
    StatementLine varDecl = node as StatementLine;

    if (varDecl != null && varDecl.LocalVariableDeclaration != null)
        foreach (Variable var in varDecl.LocalVariableDeclaration)
            vars.Add(var);
}

```

7.6. Stand-alone editor window

For test purposes, a special `CSharpEditor` project has been developed to test the structured editor component in an almost empty host. It is a simple dialog which hosts a `ViewWindow` and a `CodeUnitBlock` displayed in this `ViewWindow`:

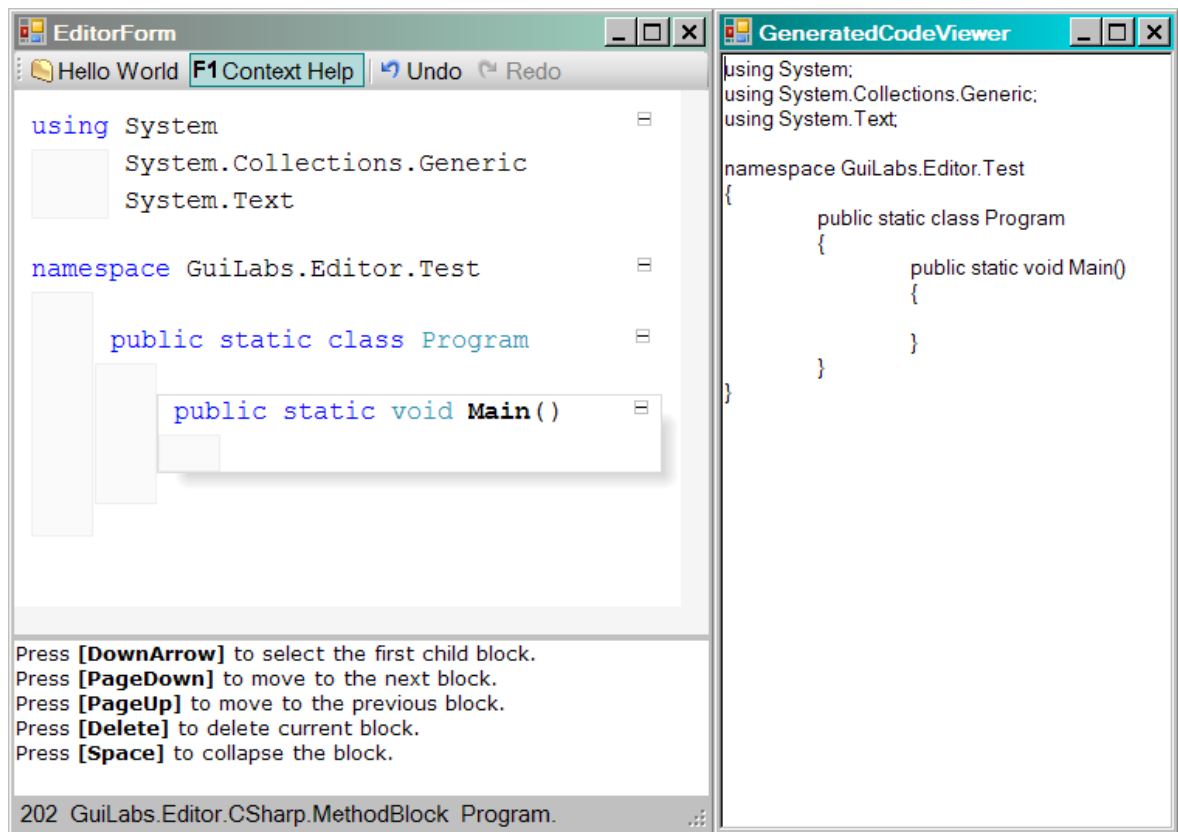


Figure 29 - CSharpEditor

This window also features the dynamic help window which displays the output of the `HelpStrings` property of the active block. The `GeneratedCodeViewer` is a window to test the functionality of the `PrettyPrinter` visitor.

8. Integrating into SharpDevelop

This chapter describes the StructuredEditor add-in created for the SharpDevelop IDE. It is an extension of SharpDevelop which allows viewing and editing .cs files in the structured editor inside the IDE.

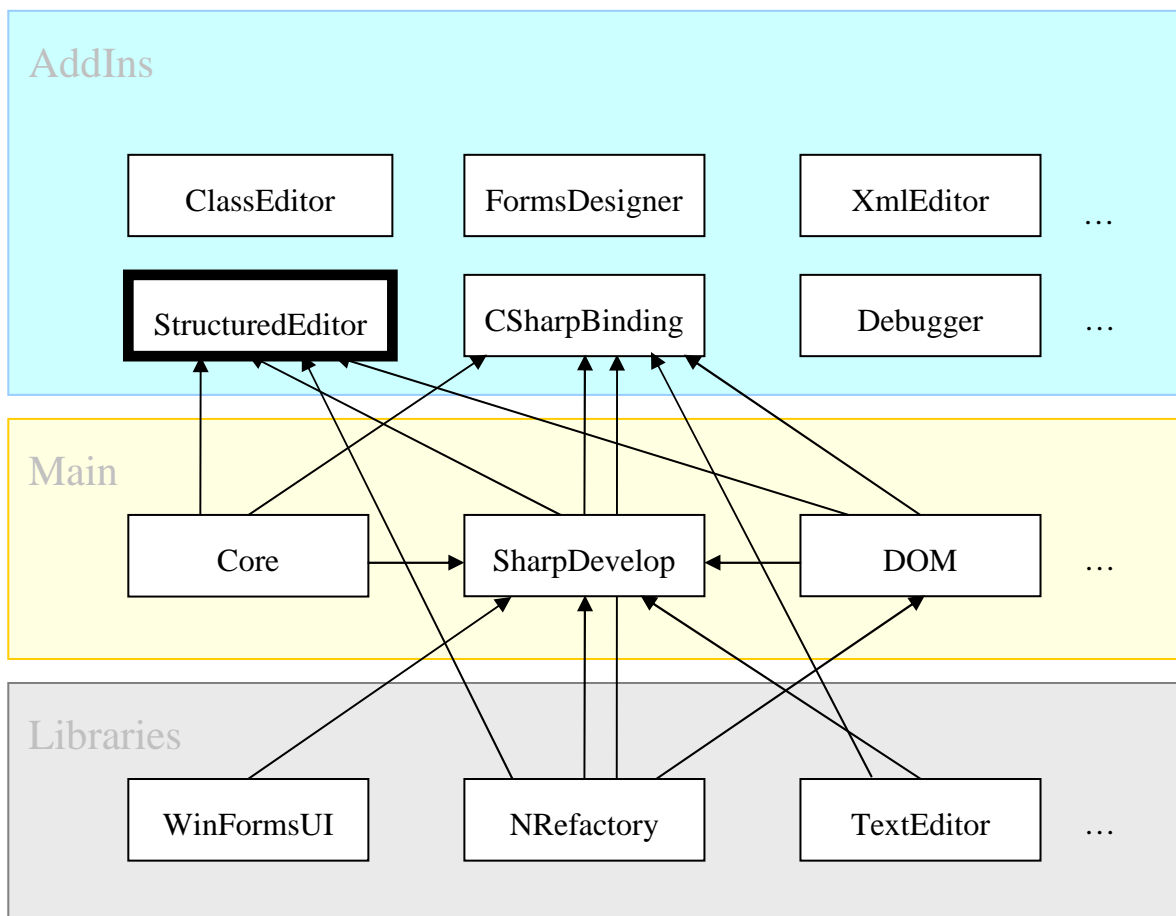
8.1. The SharpDevelop IDE

SharpDevelop ([SD]) is an open-source IDE for the .NET Framework written in C#. It was carefully chosen as the best suitable host IDE for the purposes of this work. The other candidate would be Microsoft Visual Studio 2005 ([VS]), but it's not easily extensible architecture couldn't compete with easily extensible and well-factored architecture of SharpDevelop. A problem with Visual Studio is that it doesn't currently provide easy integration scenarios for access to the parse trees, common project system and language services for third-party packages. The C# compiler isn't easily reusable and extensible and doesn't provide a clean API surface. On the contrary, SharpDevelop gives its Add-Ins very comfortable access points to all internal data structures and API surface. Final reason to choose SharpDevelop was the availability of its (fully managed) source code, which is, as always, the best documentation.

8.2. Architecture

The source code of SharpDevelop consists of 60 C# projects¹ grouped in three layers. The first layer, “Libraries”, consists of 5 independent library projects, which are used throughout the rest of the code. The second layer, “Main”, consists of 6 projects which constitute the basic IDE itself. The third layer, “AddIns”, contains pieces of functionality which add value to the IDE shell.

SharpDevelop could be described as a “microkernel” IDE. The core of the IDE is a very lightweight shell (application framework), which provides almost no user functionality, except for means to easily integrate custom functionality provided in external packages called add-ins. All functionality pieces, such as the debugger, component browser, C# language service, forms designer etc. are all provided as add-ins, which can be plugged into the IDE by simply copying files into the AddIn folder.



¹ This and other references to SharpDevelop imply the version from May 6, 2007, SharpDevelop 3.0, revision 2510

8.2.1. NRefactory

NRefactory is the most relevant library for our purposes: it defines the common AST data structure, as well as scanners, parsers, pretty-printers and other visitors for the C# and VB.NET languages. We will only use the C# part of it, although the VB.NET part of it is implemented just as well.

The scanner (called here “lexer”) and the parser are being generated using the Compiler Generator Coco/R¹. The C# grammar is described in an attributed grammar file (cs.atg). The parser produces an AST tree as its output, with the root in an instance of the `ICSharpCode.NRefactory.Ast.CompilationUnit` class. All nodes of the AST (including the compilation unit itself) inherit from the `AbstractNode` class and implement the `INode` interface.

When we talk about the AST, it is important to distinguish two totally different trees. First, an inheritance tree describes what AST **classes** inherit from what other AST classes (at design time). Second, an object tree describes what AST node **objects** are contained within a given AST node object (at runtime). These two trees may not necessarily coincide. The grammar specifies, what node objects can be contained within what node objects.

Here are some classes to demonstrate how the AST is modeled at design time:

- `NamespaceDeclaration` class models a namespace. It inherits directly from the `AbstractNode` class and introduces a string property for the namespace name.
- `TypeDeclaration` represents a declaration of any of the 5 .NET types. It could be a class, a struct, an interface, an enum or a delegate.
- `TypeReference` represents any type, which could be return type of a method or a type of a variable. This includes array types and generic types.
- `MethodDeclaration` and `PropertyDeclaration` model methods and properties correspondingly.

¹ Copyright (c) 1990, 2004 Hanspeter Moessenboeck, University of Linz, Austria; extended by M. Loeberbauer & A. Woess, Univ. of Linz with improvements by Pat Terry, Rhodes University

Also, the NRefactory library provides a set of visitor classes, which represent operations on the AST data structure that are implemented separately (see also the `Visitor` design pattern). We will need to implement our custom visitor that walks an AST tree and builds a tree of blocks out of it.

8.2.2. ICSharpCode.SharpDevelop.Dom

SharpDevelop maintains information about all types and members declared in the opened project as well as referenced projects. This information is mostly used for the features of the IDE that make it “intelligent” (code-aware). The most important one is code completion: displaying a context-aware drop-down list of types and type members directly as the user is typing.

The type information for each project is stored in special data structure called DOM (document object model). DOM is another important component of the IDE, which resides within the Main layer. The main purpose of DOM is to support code completion and to share project information between different components of the IDE.

The DOM is another tree-like data structure like the NRefactory AST, but there are some differences. The DOM is used to model only higher-level (interprocedural) language constructs, like class, member, parameter (it doesn’t cover statements and expressions). Another difference is in the fact that the DOM tree is resolved (type references actually link to type declarations), whereas the AST tree is unresolved. This is because an AST is created as a stand-alone file and the parser doesn’t know about other types and declarations. The DOM, on the contrary, is created in context of the project and knows about the project content.

This peculiarity explains why the DOM is used as the central data structure throughout the SharpDevelop IDE: it is a greatest-common-divisor that binds together different components. A DOM tree can be produced from many different sources. The first is a source file, which is being parsed to produce the AST. Then, a special `NRefactoryASTConvertVisitor` converts the AST to DOM. While being converted, the tree is actually being resolved, i.e. type references are being introduced to type declarations.

Another DOM source is reflection. It extracts type and member information from referenced assemblies. We will create a DOM structure out of the blocks hierarchy to

provide the rest of the SharpDevelop with information about what classes, members and parameters have been defined inside our structured editor.

We use the AST to **input** data from the outside world into our structured editor. The existing text source is parsed and the AST is used to create the blocks tree.

On the contrary, we use the DOM to **output** data from the structured editor to the outside world, to tell other components what have we defined within our structured editor.

8.2.3. Background compiler

Normally, when editing a program using the SharpDevelop text editor, a background compiler runs in a separate thread to keep the DOM up to date. The background compiler parses the current text buffer to obtain an AST tree and applies `NRefactoryASTConvertVisitor` to build a DOM tree out of it. A DOM tree has its root in an object of type `ICSharpCode.SharpDevelop.Dom.ICompilationUnit`.

We'll call "compilation unit" the DOM `ICompilationUnit`, and not the AST `CompilationUnit`.

It is worth mentioning, that for each opened file two compilation units are maintained: the valid compilation unit and the most recent compilation unit. When the user is currently editing the text buffer, the contents of the file may be syntactically incorrect, and as a result the background compiler cannot retrieve information about the file being edited (compile errors). Nevertheless, when the user is editing some other file, he/she can still use the constructs defined earlier in the valid compilation unit. Thus, each valid compilation unit is always older than the most recent compilation unit. When the contents of the file can be parsed without errors, both compilation units are the same.

The background parser was implemented in SharpDevelop to retrieve information about the program being edited in the background. Parsing the program is a lengthy operation and would considerably slow down editing if the program needed to be reparsed after every key pressed. To let the user type quickly (without waiting for the file to be parsed), a background thread is used, which parses the file in-between user edits. The advantage of this is that the user can type as fast as she/he wants without waiting for the parsing to

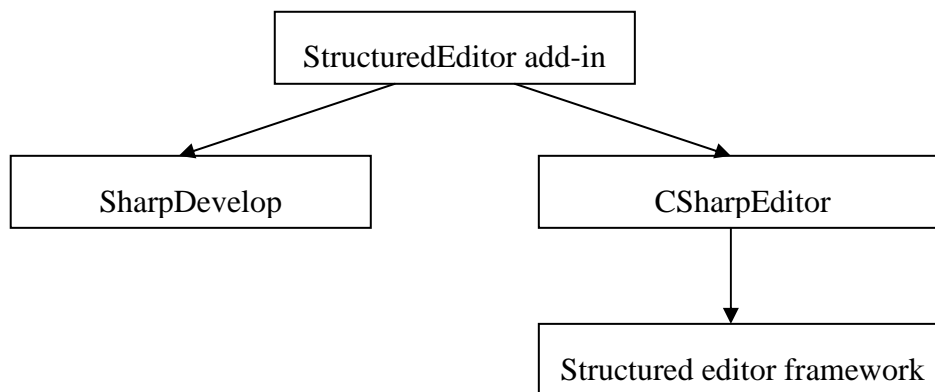
finish. The disadvantage is that the project information (DOM) is always some seconds old during typing, and may not reflect recent changes introduced by the user.

8.3. SharpDevelop add-in

Now we describe the means by which the structured editor component was integrated into SharpDevelop. We have defined the StructuredEditor project (a C# .dll) within the “AddIns” layer of the SharpDevelop source code.

An add-in can be designed without having access to the SharpDevelop source, but it is convenient to have both in the same solution.

The add-in project depends on both SharpDevelop and the structured editor component described in previous chapter. It is noteworthy that neither SharpDevelop depends on the structured editor, nor the structured editor depends on any of the SharpDevelop components – they are both totally independent of each other. Thus, the StructuredEditor add-in serves as glue that binds together SharpDevelop and the StructuredEditor, so that they interoperate with each other:



The only file that distinguishes the project from any other .dll is the StructuredEditor.addin file in XML format. It contains instructions for the SharpDevelop core how to load the addin and what parts of the IDE it defines:

```
<AddIn
  name          = "StructuredEditor"
  author        = "Kirill Osenkov"
  url           = "http://www.osenkov.com/diplom"
  description   = "A structured source code representation">

  <Runtime>
    <Import assembly = "GuiLabs.Editor.AddIn.dll"/>
  </Runtime>

  <Path name = "/SharpDevelop/Workbench/DisplayBindings">
    <DisplayBinding
      id = "StructuredEditorBinding"
      type = "Secondary"
      class = "GuiLabs.Editor.AddIn.StructuredEditorDisplayBinding"
      fileNamePattern = "\.(cs)$" />
    </Path>
  </AddIn>
```

The most important part is the name of the assembly .dll file to load. Also, it specifies that a new display binding is added to every .cs file. A display binding is a representation for documents. Every document in SharpDevelop can have many pages (windows) associated with it. There is one primary display binding and possibly many secondary display bindings. Our structured editor will be a secondary display binding which uses the normal C# plain text editor as the primary display binding. Visually, it will look like the following:

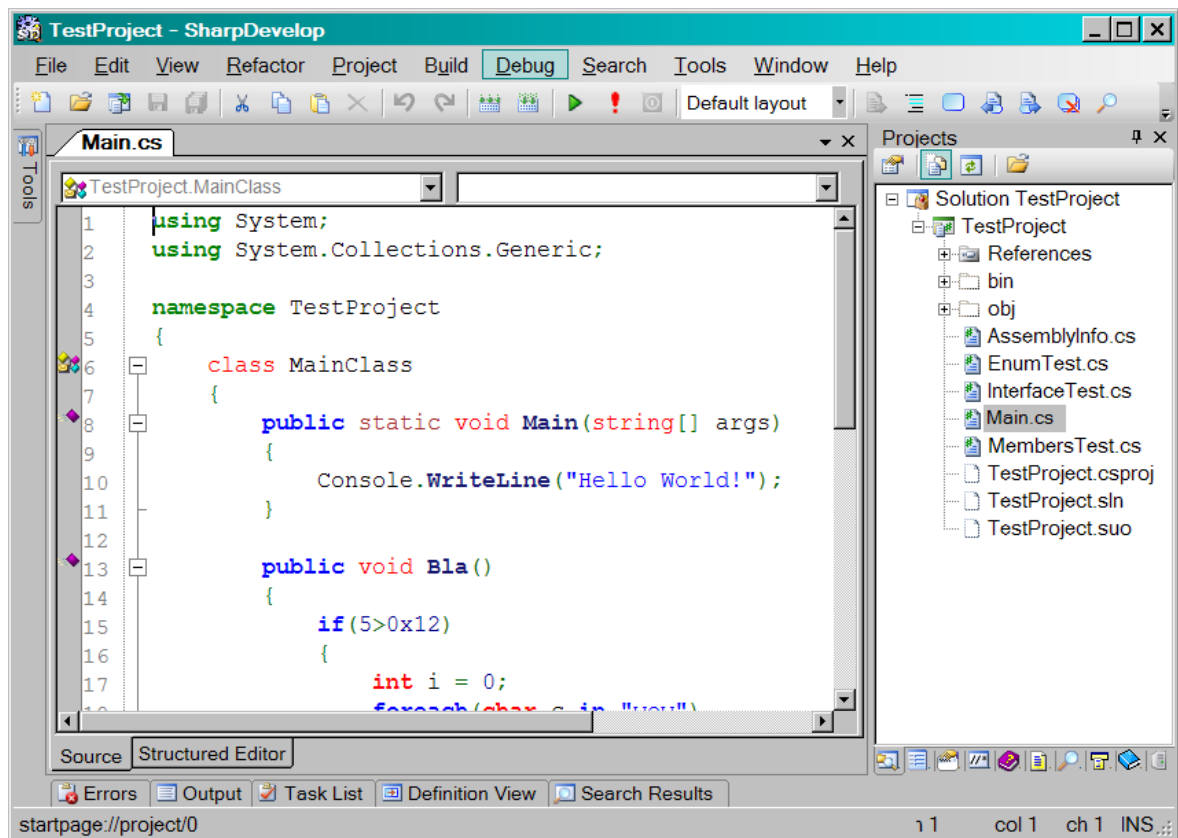


Figure 30 - C# display binding (primary)

The main document window in the middle of the IDE is the primary C# view of the Main.cs file. Below this window there are two tabs: Source and Structured Editor. Source displays the primary C# binding, while Structured Editor displays our integrated structured view of the same program:

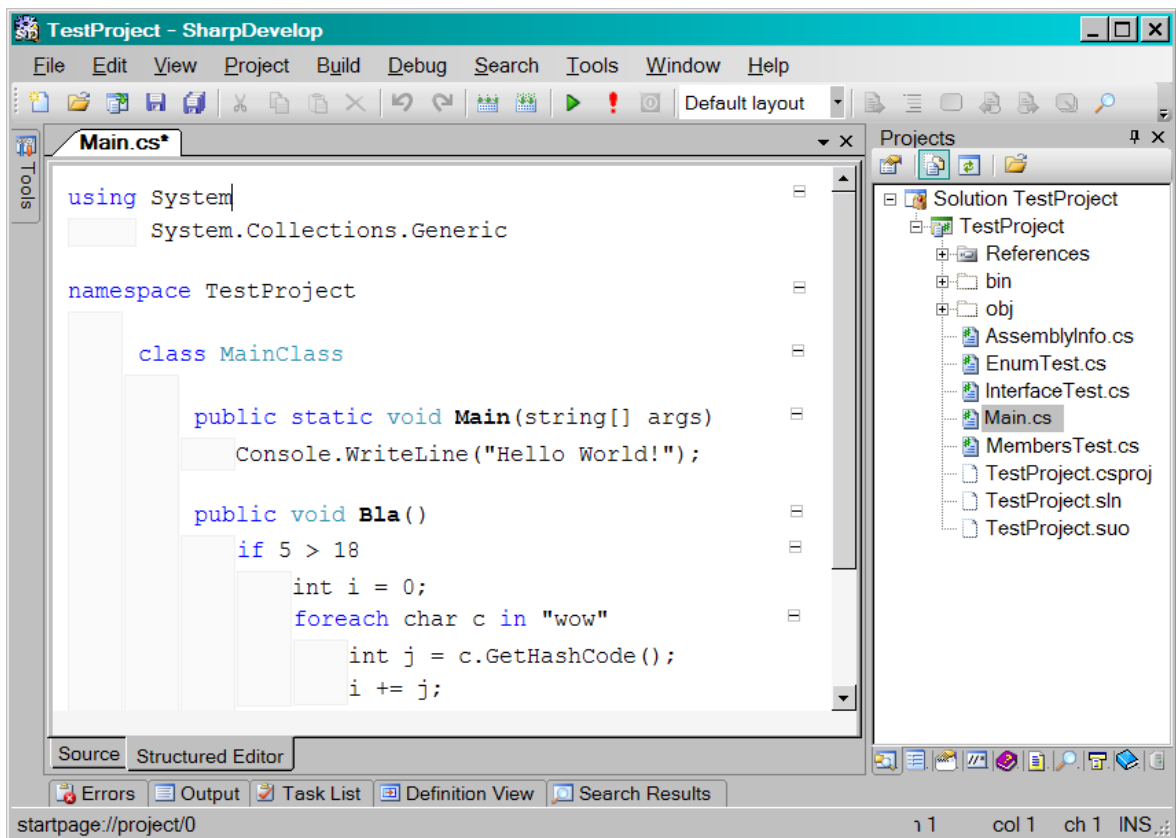


Figure 31 - Structured Editor display binding (secondary)

8.4. Round-tripping

In the moment when we switch from the Source to the Structured Editor view, the following happens. First, the add-in parses the source code of the document being edited and produces the AST out of it. Then, a special `BlocksFromASTVisitor` walks the AST tree and builds the tree of blocks out of it (with `RootBlock` as the root). Then, this `RootBlock` is loaded into the `ViewWindow` which is hosted on the surface of the secondary display binding.

Every time the user edits the block structure (after every transaction is being recorded by the `ActionManager` of the `RootBlock`), the `DomFromBlocksBuilder` visitor walks the blocks structure and produces the DOM tree out of it. The DOM tree is sent to the SharpDevelop project service to notify the environment of the changes we have made. Thus, when the user opens some other file in the same IDE, it will be possible to use constructs just defined in our structured editor. These constructs will appear in the completion list at the right place.

It is important to note, that no background thread is used in this case to provide SharpDevelop project service with the current DOM of the file. The conversion of blocks to DOM is fast enough to happen in realtime in the main thread directly after each user edit. That is why all the information about types and members defined in the structured editor is delivered to SharpDevelop immediately and is always up to date. Thus, we preserve the advantage of the text editor (the user can type fast without waiting for the parser to finish) and get rid of the disadvantage (unlike with the text editor, the DOM is always up to date now). This is an important advantage of the structured editor caused by the fact that the blocks tree is already “pre-parsed” so most of the parser’s work is already done. And, there is no need to maintain two compilation units for one file.

Another advantage of this approach is that even if some language constructs are in a syntactically incorrect state (e.g. name missing), the method will still correctly return valid information about the remaining types and members. A parser-based approach would not be able to finish parsing of the remaining constructs once it encountered a syntax error. In this respect, a structured editor is more stable and robust than a text editor.

When the user switches back from the structured editor view to the primary source view, the `PrettyPrinter` visitor runs over the blocks tree and outputs the new automatically formatted text of the program. This text replaces the old text which was there before we opened the structured editor.

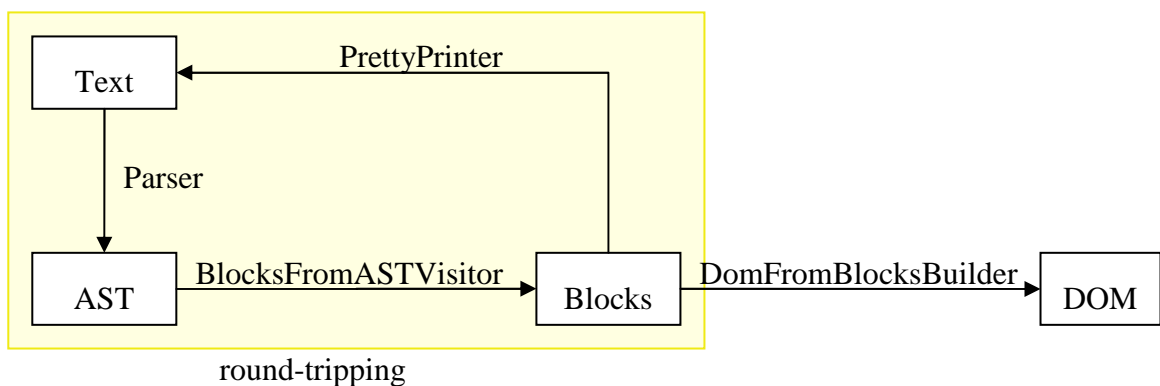


Figure 32 - conversion of data structures

The process of converting text to blocks and back is called round-tripping. Unfortunately, the current implementation does not preserve formatting and comments of the source text

during round-tripping. Each time when the structured editor is opened and closed, the text of the program is being regenerated without considering whitespace and comments.

It is possible (and even straightforward) to implement formatting-preserving round-tripping process. The NRefactory parser and AST even provide useful means for storing comments and whitespace in the AST tree (called „tracking specials“). However the overall complexity of this work didn't leave resources to implement formatting preserving. The author understands that formatting and comments are crucial for all developers, and losing it is really unacceptable. However, the editor being presented is just a proof-of-concept and is currently not intended for use in a production environment.

As switching between two display bindings is already a very expensive process in the SharpDevelop IDE, the additional performance penalty of the round-tripping process introduced in this thesis is negligible.

8.5. Implementation of the add-in

8.5.1. StructuredEditorDisplayBinding

StructuredEditorDisplayBinding is a class, which serves as the binding point of the editor control inside the SharpDevelop IDE. This class is mentioned as the binding class in the StructuredEditor.addin configuration file. When the add-in is loaded, SharpDevelop core instantiates this class and associates it as a secondary display binding for all files with the extension “.cs”. This means, as soon as any .cs file will be opened in the SharpDevelop editor (which is the primary display binding), an additional tab will be displayed underneath the main editor view. When the user clicks this tab, a new view of the file will be opened.

The secondary window, where the structured program view is displayed, is provided by the StructuredEditorViewContent class. As soon as the secondary view is opened by the user, the

```
StructuredEditorDisplayBinding.CreateSecondaryViewContent()
```

method returns an instance of the StructuredEditorViewContent class.

8.5.2. StructuredEditorViewContent

This class is the main part of the StructuredEditor add-in. Its responsibilities include:

- Converting the program text from the primary display binding to the blocks tree of the secondary display binding (parsing). This is done in the `LoadFromPrimary` method.
- Converting the blocks tree back into text (of the primary text editor) – done by `SaveToPrimary` method.
- Providing the `ViewWindow` control to place in the SharpDevelop window, where all the blocks will be drawn.
- Hosting the `CodeUnitBlock` object, the root of the blocks hierarchy.
- Providing the `RecreateDom` method, which updates the DOM information of the current file, thus letting the rest of the SharpDevelop know, what types and members have been defined in the structured editor.
- Providing the `LanguageService` class to the block tree. This class is mostly used to give the editor the necessary external information it requires. For example, the language service gives the editor means to parse text fragments. As the editor is a hybrid between pure structured editors and text editors, it displays some fragments in plain text. The language service is used to parse these text fragments, to extract information out of them and to provide code completion information for them.
- Providing the code completion for statements. As statements are stored as plain text, it is necessary to parse them and to be able to show code completion as the user types. This is done by the `PrepareCompletionForStatement` method.

8.5.3. CSharpLanguageService

As the `CSharpBlocks` component described in the previous chapter does not know about where it is hosted (in this case, it does know nothing about SharpDevelop or its add-ins), it requires some means of communicating with its hosting environment. This functionality is provided by the `LanguageService` class. It serves as a “façade” for the editor to communicate with its (unknown) environment.

Every `RootBlock` has a reference to an object of type `LanguageService`. If it is null, the editor assumes it is not bound to any hosting environment, and many of the functionality (code completion, etc.) is disabled.

The `StructuredEditor` add-in defines the class `CSharpLanguageService`, which inherits from the `LanguageService` class and provides everything the editor wants to know about its environment. For example, when method parameters are entered in the editor, the editor itself doesn't know what parameters are they, because the editor doesn't have a parser (parameters are stored as a text string). The class `CSharpLanguageService` provides the parser for parameter strings, so every time the parameters string is changed, the editor asks `CSharpLanguageService` to parse it and give back the detailed information about the parameters. Now the editor knows what parameters are inside that string.

Here is a list of functionality provided by the `CSharpLanguageService`:

- Providing the `CSharpParserService`, a helper class to parse text fragments stored in the editor and to provide the editor with information what is written in those text fragments.
- Providing a list of namespaces available within the current namespace. This feature makes use of the `SharpDevelop` parser service and project contents, where the DOM information is stored.

8.5.4. `CSharpParserService`

`CSharpParserService` provides methods to convert strings to parts of the AST.

Normally, the compiler API should provide this functionality. But the current C# compiler being used (Microsoft `csc.exe`) does not provide an API surface of required granularity.

For example, it provides a method to create a parsed list of parameters (`ParameterList`) out of a simple string (e.g. `"int a, string b, params object[] objects"`). The implementation of most such methods in `CSharpParserService` is based on the following steps:

1. Surround the input string by a dummy class declaration (the example above could be surrounded by “`class A{void B(%){} }`”, where % is the parameter string from above).
2. Parse the whole string as if it was an independent compilation unit.
3. Find the node in the resulting AST tree which corresponds to the source string and return it.

In future versions of the structured editor, it might be possible to implement parameter lists not as a text string, but in a structured manner (just like the access modifiers are implemented as blocks, and not as plain text).

Samples of such structured lists of words already appear in different software systems. For example, many e-mailing programs such as Microsoft Outlook allow to edit the list of e-mail addresses (in the To: field) in a structured way: an e-mail address can only be selected as a whole, the caret can select the entire e-mail address at once, the caret considers it as a single character and jumps over it. However, more usability research has to be conducted in this innovative area of user interaction, which cannot take place within the scope of this thesis.

The editor also uses the `CSharpParserService` to parse statements. As some statements could actually contain a variable declaration (“`int i = 0`”), it is necessary to extract names and types of all variables defined within the current statement. Once a statement has been parsed by the `CSharpParserService.ParseStatement` method, an instance of the `CSharpStatement` class is created for the statement block being parsed. This `CSharpStatement` contains information whether this statement is a variable declaration, and if yes, provides that name and type of the variable. This information is being used later by the editor to show the list of local variables in the completion list for other statements.

8.5.5. BlocksFromASTVisitor

This is a typical implementation of the Visitor design pattern. It is used to walk the AST tree created by the `NRefactory` parser and to build corresponding blocks inside the `CodeUnitBlock`.

The `AST CompilationUnit` corresponds to the `CodeUnitBlock`. The `NamespaceDeclaration` becomes `NamespaceBlock`. A `TypeDeclaration` can become a `ClassBlock`, a `StructBlock`, an `InterfaceBlock`, an `EnumBlock`, etc.

8.5.6. DomFromBlocksBuilder

The `DomFromBlocksBuilder` class works in the opposite direction: it walks the blocks tree and creates a DOM tree out of it. As the DOM only needs information about the higher language level (interprocedural, i.e. types and members), the bodies of methods and properties are being ignored.

A `DOM ICompilationUnit` is being created from the `CodeUnitBlock`, a `DefaultClass` is being created from a `ClassBlock`, etc.

8.5.7. BlocksToDomMap

An important thing to know about conversion blocks to DOM is the following. `SharpDevelop` (especially the DOM structure) doesn't know about the blocks classes, and the blocks classes do not know about the DOM structure either. However, to be able to work with code completion, we need to know, what DOM object describes what block in our editor. Of course, one could introduce a direct reference on the `Block` class: `Block.MyDOMObject` could just keep a reference to a corresponding DOM object. However, we wanted to keep the blocks hierarchy independent of `SharpDevelop`, so that the editor component could also be integrated into other IDEs in the future. This maintains the flexibility of the architecture.

That is why an external map is introduced: `BlocksToDomMap`. It is actually a hashtable, which keeps pairs (`Block`, `DOMObject`). If we want to find out, what DOM object corresponds to some block, we just do a lookup in this hashtable and the necessary object is returned, if it exists. Otherwise, `null` reference is returned.

8.6. Code completion

8.6.1. DotCompletion

`DotCompletion` is a helper class used to show completion list when the user types in a dot to show types and type members. Completion is mostly being used inside a `StatementLine` block, which represents a statement inside a plain text box.

`StructuredEditorViewContent` creates a new object of type `DotCompletion` and calls its `Prepare()` method:

```
public void Prepare(StatementLine statementBlock)
{
    string expression =
        statementBlock.MyTextBox.TextBeforeCaret.TrimEnd('.');

    ExpressionResult exprResult =
        GetExpression(expression);

    ResolveResult resolveResult =
        GetResolveResult(exprResult);

    ArrayList completionData =
        GetCompletionData(resolveResult);

    AddResolveResults(
        statementBlock,
        completionData,
        exprResult.Context);
}
```

This method does the following:

1. The text in the statement textbox before the caret is extracted using the `TextBeforeCaret` helper property. Because the user has already entered the dot, it must be truncated from the right part of the string.
2. The least complete expression is extracted out of this text using a mini reverse scanner and parser. This is necessary to correctly determine an object, to which the dot operator should apply. Consider, for example, the following string: `this.SomeProperty.GetSomeOtherObject(2+3)`. After this dot, the code completion engine should show members of the type which is returned by the `GetSomeOtherObject` method. The `ExpressionFinder` class of `ICSharpCode.SharpDevelop.Dom` project is used to extract the least complete expression out of a text string.
3. The calculated expression is resolved – the return type of the expression is calculated. This process involves the `Resolver` class and the `TypeVisitor` class, which are discussed later.
4. The list of appropriate namespaces, types, members, parameters and local variables is determined, based on the return type of the expression.
5. The list of completion data is added to the completion list which is about to be shown for the statement textbox.

8.6.2. Resolving

An important part of the code completion process is the `Resolver` class. The `Resolver` is used to determine the return type of a given text expression.

`SharpDevelop` already defines a class called `NRefactoryResolver`, which is used by `SharpDevelop` code completion engine to resolve the type of the expression to the left of the caret. It is tightly coupled with the `TypeVisitor` class. Unfortunately, it was not possible to reuse `NRefactoryResolver` and `TypeVisitor` to resolve string expressions from the structured editor. The reason is that both classes rely heavily on the assumption, that the source code to be analyzed comes from a text file, where the position of the caret inside that file is given in line and column coordinates.

A pair of integer coordinates which denote line and column numbers within a text file is called a *location* within the file. A pair of locations is called a *region*.

Most functionality of the standard resolver accepts two numbers (line and column numbers) to indicate position of the caret in the source code. This information is used inside the resolver to obtain the class and member where the caret currently is. This is achieved by the following steps. An AST contains beginning and ending locations for each node. To find the node where the caret currently is, the resolver iterates over all nodes in the AST and compares if the caret location lies within the node region. The resolver needs this functionality to determine the current class and member where the caret currently is.

This is the reason why this functionality couldn't be reused. The structured editor doesn't have any information about text positions in code because there is no text code. There isn't even such a thing as a line of code – there are only declarations and language constructs.

This all led to adapting the code from `Resolver` and `TypeVisitor` for structured editing. The position of the caret is now represented by a special class called `PlaceInCode`. This class provides all the context information the resolver might need – including the current class and member. With a structured editor, such information is easily obtained for every block. Thus the architecture of the resolver is greatly simplified and made more generic, because the resolver doesn't rely on the fact that the code is stored as text anymore. We could now reuse it for instance, for resolving source code that comes from a database.

The resolver is using a parser to obtain the AST for the expression in question and then applies different resolving methods for each possible type of the expression found.

9. Summary

This thesis proposes a vision which anticipates an evolution of IDEs with careful and gradual accommodation and refinement of structured editors together with traditional code editors. The hope is expressed that the architecture of IDEs will be improved, providing a clean API surface to allow full interaction with the compiler, parse trees and language services. The code could actually become a model from the MVC pattern, with different views on it.

A proof-of-concept implementation of a structured editor is demonstrated, along with the structured editor framework, which serves as a foundation, and a SharpDevelop add-in, which integrates the prototype structured editor into the SharpDevelop IDE.

Despite of the fact, that the editor was not tuned for performance (it's an experimental prototype which should only demonstrate the general possibility of building such editors), the performance of the editor turned out to be quite reasonable.

Generally, more work has to be done to compare the approach and functionality presented in this thesis with existing results. More usability research has to be conducted as well. However, such comparative analysis is complex and could not become a part of this thesis because of resource limitations.

Up-to-date online information about this thesis, as well as links and errata, is maintained at the website www.osenkov.com/diplom.

9.1. Future research directions

9.1.1. A DSL for structured editors

The most often question that comes into mind when thinking about structured editors – is the language grammar hard-coded into the editor? For this work, the answer is yes. Otherwise a tremendous amount of additional work would have to be performed while losing functionality and probably performance.

However, one could really think about two more approaches to creating structured editors, apart from hard-coding the grammar into the editor.

The first alternative is to use an editor generator, which is the most common approach. Given a special domain specific language that describes such editors, and an editor generator, one could generate editors for any custom language. This path is tempting, but it hides a lot of complexity and implementation limitations. A common problem is that a lot of language constructs have to be hand-crafted because they are an exception to the rules and not behave like other constructs.

Another way, even more futuristic, is making an interpreter, which could dynamically load the editor description at runtime, just like XML editors dynamically load XSD schemas at runtime. However, the impression is that this way needs attention only after hand-crafted and specially tuned structured editors have become mainstream.

9.1.2. Static analysis

A lot of static analysis and code querying tools currently rely on the code stored in plain text files. They explicitly parse the source code into a specialized database to be able to extract information about the source code. With the introduction of structured editors, one could probably implement a query engine, that collects code metrics in realtime, without the need to scan the database first. Or, one could at least update the database live, without the need to rescan the code.

One common measure for software complexity is the number of lines of code. With structured editors, other measurements may be possible, for example, the number of used language constructs, which could be more precise. Different code metrics are possible, which take into account weighted combinations of different parameters.

9.1.3. Markup languages

A structured representation is possible not only for general purpose programming languages, but also for many markup and modeling languages, such as HTML, XML, VHDL, Haskell, Epigram, LaTeX etc.

For markup languages, we can classify the “structuredness” of an editor in three grades:

1. Text (less structured, traditional, stream of characters, possibly with additional features like code completion, collapsing, etc.)
2. Structured editors (in the middle, still represents the same low-level markup concepts as text, although on a more structured level). Structured editors allow same low-level control of every tag, but in a different way.
3. WYSIWYG are most abstract, don’t directly represent tags anymore. Instead, WYSIWYG editors provide the end-result of how the final document should look like.

Structured editors slightly differ from WYSIWYG editors in a way that structured editors (like plain text editors) display the internal structure of the document as it is whereas WYSIWYG editors display an adapted view on that structure, customized for the end-user’s perception convenience. For example, a word-processor doesn’t actually display paragraphs with all attributes (alignment, font, etc.) explicitly as boxes, but instead formats the edited document as if it were the final output of the editing process.

Most viable will probably be the hybrid editors, which take the best parts of both worlds. For example, a hybrid between text and structure is presented in this work. A hybrid between structure and WYSIWYG is great for document editing.

9.1.4. Domain Specific Languages

Another perspective for structured editors is definitely the “mini-languages” – the domain-specific languages. More research has to be done on applying structured editor technology to creating customized highly interactive mini-editors that are fully content-aware.

A common intuition states, that if a person has to edit raw XML in an XML editor, chances are that a specialized structured editor would be a better choice for this task.

9.1.5. Extending the editor vs. extending the language

Usable structured editors open a totally new perspective on the design, development and improvement of programming languages. Now it is possible to extend the language without actually extending the language. One could introduce new blocks (interactive UI elements) for new language constructs (e.g. syntactic sugar), which would be internally represented in terms of the same old language constructs. Without making the compiler backend to take care of generating code for syntactic sugar, one could just define new blocks which would emit the necessary code. One wouldn't even need to change the backend.

9.2. Drawbacks of the current implementation

9.2.1. Only a subset of C# 1.0 is supported

Unfortunately, it was not possible to completely implement a structured editor that fully supports even C# 1.0 in the scope of this thesis. But the main goal has been reached – to provide a proof-of-concept editor that solves most problems that normally arise when building a structured editor. Some functionality (such as intelligent drag & drop support) couldn't be implemented because of time constraints.

9.2.2. The necessity of round-tripping

The lack of observable AST/DOM data structures caused an architecture decision, which is not perfect. Instead of implementing the structured editor as another View on the code Model, the author was forced to implement round-tripping between blocks and text.

9.2.3. The need for CSharpParserService

CSharpParserService is a class which parses different parts of a program from text into chunks of AST. This class wouldn't have been necessary at all, if the IDE provided a clean usable compiler API surface, i.e. a set of classes, methods and properties that would allow parsing not only of whole compilation units, but also of arbitrary parts (such as parameter list). Such an API surface would greatly simplify parsing the text pieces of a hybrid editor.

9.2.4. The need for custom resolver

The implementation of the resolver provided by SharpDevelop (`NRefactoryResolver` and `TypeVisitor`) has a serious limitation. It was designed with text-based source code in mind. That is why a great deal of the API surface operates on such information as line and column numbers. The lack of an abstract resolver was the reason to refactor the `NRefactoryResolver` and `TypeVisitor` classes to be independent of line and column information, which turned out to be a difficult challenge.

10. References

- [AhoSeUl] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: Compilers: principles, techniques & tools. Second Edition, 2007
- [Amaya] Amaya, <http://www.w3.org/Amaya> - WYSIWYG and structured browser and editor
- [BoxView] BoxView – The multiview project at Portland State University, http://multiview.cs.pdx.edu/refactoring/statement_view
- [CzEi] Krzysztof Czarnecki and Ulrich Eisenecker: „Generative Programming: Methods, Tools, and Applications“. Addison-Wesley, Reading, MA, USA, June 2000 – Chapter 11 (Intentional Programming)
- [Eclipse] Eclipse, <http://www.eclipse.org>
- [ECMA] Standard ECMA-334: C# Language Specification, 3rd Edition, June 2005
- [FDG] Krzysztof Cwalina, Brad Abrams: „Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries“, Addison-Wesley Professional, 2005
- [Fowler] Martin Fowler: „Language Workbenches – a Killer App for Domain Specific Languages?“, <http://martinfowler.com/articles/languageWorkbench.html>

- [Fowler2] Martin Fowler: “Language Workbenches in Action – MPS”,
<http://martinfowler.com/articles/mpsAgree.html>
- [FxCop] FxCop: Microsoft Source Code Analysis Tool,
<http://www.gotdotnet.com/team/fxcop>
- [GoF] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Design Patterns: Elements of Reusable Object-Oriented Software“, Addison-Wesley Professional
- [Grammar] Hyperlinked ECMA C# Grammar, http://www.jaggersoft.com/csharp_grammar.html
- [HIP] Microsoft Research, Human Interactions in Programming group,
<http://research.microsoft.com/projects/hip/>
- [IntelliJ] JetBrains IntelliJ IDEA, <http://www.jetbrains.com/idea>
- [IntentSoft] Intentional Software Corporation, www.intentsoft.com
- [IntentSoft2] Charles Simonyi, Magnus Christerson, Shane Clifford, “Intentional Software”, an OOPSLA 2006 paper,
http://www.intentsoft.com/technology/IS_OOPSLA_2006_paper.pdf
- [IP] Charles Symonyi et. al. „Intentional Programming“,
http://www.intentsoft.com/technology/IS_OOPSLA_2006_paper.pdf
- [Lava] Lava: an experimental object-oriented rapid application development (RAD) language with parameterized ("virtual") types, refactoring, and extensive static checks. <http://lavape.sourceforge.net>
- [LOP] Sergey Dmitriev: „Language Oriented Programming: The Next Programming Paradigm“,
<http://www.onboard.jetbrains.com/is1/articles/04/10/lop/>
- [LutzR1] Lutz Roeder: „Interactive Source Code“,
<http://aisto.com/roeder/Paper/InteractiveSourceCode.ppt>
- [LutzR2] Lutz Roeder: „Extensibility and Visualization of Source Code Documents“,
<http://www.aisto.com/Roeder/Paper/SourceCodeDocuments.ppt>

- [LutzR3] Lutz Roeder: „Transformation and Visualization of Abstractions using the IP System“, <http://www.aisto.com/Roeder/Paper/IntentionalProgramming.ppt>
- [Mozart] Concept Programming vs. Intentional Programming, <http://mozart-dev.sourceforge.net/cp-vs-ip.html>
- [NDepend] NDepend, <http://www.ndepend.com>
- [Nemerle] Nemerle programming language, www.nemerle.org
- [NStatic] NStatic, http://wesnerm.blogs.com/net_undocumented/2007/02/nstatic_present.html
- [POS] Eisenecker, Ulrich W.; Roeder, Lutz; Czarnecki, Krzysztof: Programmieren ohne Sprache. iX Magazin für professionelle Informationstechnik, pp. 142-147, November 2000.
- [ProgrTree] Alexander Yurov, ProgramTree, <http://www.programtree.com>
- [ReSharper] JetBrains ReSharper, <http://www.jetbrains.com/resharper>
- [ReTe1] Thomas W. Reps, Tim Teitelbaum: „The synthesizer generator: a system for constructing language-based editors“, 1989 - Springer-Verlag New York
- [ReTe2] Thomas W. Reps, Tim Teitelbaum: „The synthesizer generator reference manual“, 4th Edition, 1992 - Springer-Verlag New York
- [SCID] Roedy Green: „Source Code in Database“ (Java Source Code SCID-style browser/editor), <http://mindprod.com/projects/scid.html>
- [SD] SharpDevelop, <http://sharpdevelop.net>
- [Semmle] Semmle code querying software, <http://www.semmle.com>
- [Simonyi1] Code Generation Network: Interview with Charles Symonyi, http://www.codegeneration.net/tiki-read_article.php?articleId=61
- [SoftFact] Jack Greenfield and Keith Short, with Steve Cook and Stuart Kent: “Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools”, Wiley, 2004
- [SlickEdit] SlickEdit, <http://www.slickedit.com>
- [Subtext] Jonathan Edwards: Subtext, <http://www.subtextual.org>

- [VAssist] Whole Tomato Visual Assist X, <http://www.wholetomato.com>
- [VS] Microsoft Visual Studio, <http://msdn.microsoft.com/vstudio>
- [WesnerM1] Wesner Moise: „Whidbey may miss the next coding revolution“, http://wesnerm.blogs.com/net_undocumented/2004/06/whidbey_may_mis.html
- [WesnerM2] Wesner Moise: „Graphical Source Code Editors“, http://wesnerm.blogs.com/net_undocumented/2004/06/graphical_sourc.html
- [WPF] Windows Presentation Foundation, <http://wpf.netfx3.com>

All web links were accessed on May 12, 2007.

11. List of figures

Figure 1 – layered architecture of an IDE.....	9
Figure 2 - compiler as a black-box for the IDE	10
Figure 3 – the editor as a black-box for the IDE	11
Figure 4 – round-tripping between the AST and IDE components through code	15
Figure 5 – a structured editor directly operates on the AST	16
Figure 6 - a possible architecture of an IDE built around the AST	16
Figure 7 – approaching the ideal editor from different sides.....	26
Figure 8 - the Hello World program in the structured editor.....	36
Figure 9 - completion list inside an empty block.....	38
Figure 10 - adding a using declaration.....	41
Figure 11 - exiting from the using declaration	41
Figure 12 - completion list for creating types	41
Figure 13 - container block sample.....	42
Figure 14 - selecting a field	46
Figure 15 - statements as text	47
Figure 16 - example of a for loop	48

Figure 17 - foreach block.....	49
Figure 18 – Dependencies of an editor from the framework.....	53
Figure 19 - a block with its siblings, parent and children.....	61
Figure 20 - an example of a UniversalBlock	70
Figure 21 - sample history of actions (two actions have been executed so far)	82
Figure 22 - Action2 was undone.....	82
Figure 23 – Redo buffer is discarded.....	83
Figure 24 - parallel hierarchies of blocks and controls.....	88
Figure 25 - completion list of an EmptyNamespaceBlock	93
Figure 26 - defining a type incrementally.....	94
Figure 27 – block classes that model type declarations.....	96
Figure 28 - type member blocks	97
Figure 29 - CSharpEditor.....	104
Figure 30 - C# display binding (primary).....	112
Figure 31 - Structured Editor display binding (secondary)	113
Figure 32 - conversion of data structures.....	114